修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報理工学　学研究科　情報・通信工学専攻　博士前期課程		
氏　　名	白澤　孝仁	学籍番号	１２３１０４９
論 文 題 目	村田法のスレッド並列化によるマルチコア CPU 上での 実対称帯行列帯幅縮小操作の高速化		

要　　旨

本論文は固有値計算における帯幅縮小操作において、高いデータ再利用効率を持つ村田法を複数の同期方式を用いて並列化した際の性能を比較し、特性についてまとめたものである。

固有値計算は基礎的な数値計算であり多くの分野で使用されている。反復計算の内部で用いられることもあり、高速化は重要な課題である。一般的に固有値計算は三重対角行列を経由することで高速に計算を行う。Bischofらは三重対角化操作を帯行列化と帯幅縮小の二段階に分けることでデータ再利用性が高い行列-行列積を用いて実行する手法を提案し、現在主流の方法となりつつある。

村田法はHouseholderを用いた帯幅縮小操作である。反復計算により変換を行い、反復の一部過程において計算範囲が重複しないため、並列に実行することが可能である。しかしながら、村田法の並列実行は頻繁にデータの同期処理が発生する。同期処理はコストが高く、性能に影響を与える可能性がある。そのため同期回数が異なる３種類の同期方式に基づき並列実装を行い、性能の比較を行った。また性能差が発生した場合の原因として考えられるキャッシュミス回数と同期処理回数、全体に占める同期処理の割合を測定することで原因の考察を行った。

Intel Core i7 2600Kプロセッサ(4コア)上で性能を測定したところ、論文中で典型的なベンチマーク行列として扱っている行列サイズ10240、帯幅96において、村田法による縮小操作はLAPACKによる帯幅縮小操作より1.89倍高速化した。さらに村田法のスレッド並列実行を行うことで、8スレッド並列実行を行った場合は1スレッド逐次実行の場合より3.93倍高速化を行うことができた。また同期方式の違いにより4スレッド並列実行の時実行速度に1.12倍の差が発生し、8スレッド並列実行の時2.15倍差が発生した。一部同期方式では特定のスレッド数で同期処理が頻発し、低速化する事態が発生し、これらの結果から同期方式と実行時間の間に明確な相関があることが確認された。主な要因としてキャッシュミスによるペナルティが考えられ、性能分析からも関係が裏付けられている。また同期回数と同期処理の割合から、同期回数よりも、同期処理単体のオーバヘッドが性能に影響を与えることがわかった。

今後の課題として本研究を反映させ、並列性を最大に高めた調歩方式による同期の実装、より大きな行列に対する計算を行うため並列化方法を変更したMPIによるプロセス並列化の適用が考えられる。

2013年度修士論文

村田法のスレッド並列化による マルチコアCPU上での実対称帯 行列帯幅縮小操作の高速化

2014年3月10日

指導教員 岡本 吉央 准教授
緒方 秀教 教授

電気通信大学情報理工学研究科
1231049　　白澤 孝仁

目次

第 1 章　はじめに

1.1　背景

実対称行列の固有値計算は基本的な数値計算の一つであり，分子軌道計算，構造物の振動解析，統計処理など幅広い分野で使用されている [1]．計算した固有値を元に行列を修正し，繰り返し固有値計算を行うことも多いため固有値計算の高速化は非常に重要な課題となっている．

近年プロセッサ性能の向上によりプロセッサ処理速度とメモリ通信速度の間に著しい差が発生しており，プロセッサ-メモリ間の低速な通信が計算速度向上の主要なボトルネックとなっている．その解決法としてプロセッサとメモリの間に高速だが小容量のキャッシュメモリを設置する方法が存在する．したがって現在のプロセッサ環境において高い性能を引き出すためには，キャッシュの有効活用を行う, キャッシュ効率の良いアルゴリズムを使用する必要がある．

現在使用されている主なプロセッサとしてインテル社の Core i, Xeon, AMD 社の A, Opteron，IBM 社の Power, オラクル社，富士通社らの SPARC などが存在し，いずれも複数のプロセッサコアを搭載している．複数のコアを持つ CPU(以下マルチコア CPU) を使用した環境では，各コアに計算を割振り並列に計算することでプロセッサの性能を引き出し高速に計算を行うことができる．並列化を行った場合，並列化処理によるオーバーヘッドが発生する．並列化処理に伴うオーバーヘッド (以下並列化オーバーヘッド) は性能低下の要因となるためできる限り小さくする必要がある．

これらのことから高速計算のために優れたキャッシュ効率と並列性を持ち，なおかつ並列化オーバーヘッドは小さいプログラムが必要とされている．本論文では，固有値計算アルゴリズムの一部であり優れたキャッシュ効率を持つ村田法の並列化と，その際発生する並列化オーバーヘッドが小さい並列化手法について検討を行った．

第 2 章　予備知識

2.1　BLAS

ベクトル-ベクトル積, 行列-ベクトル積や行列-行列積など行列, ベクトルに関する基本線形演算は BLAS (Basic Linear Algebra Subprograms) と呼ばれるライブラリを使用して実装を行うことが出来る [2]．BLAS はキャッシュ効率を高めるためデータの再利用を行うなど，高速計算を行うために最適化されたライブラリである．各環境に合わせて最適化された BLAS を使用することで，高速化のために必要な計算順序やデータ再利用を意識することなく簡単に高速計算を行うことができる．

BLAS にはベクトル-ベクトル演算を行う Level 1, 行列-ベクトル演算を行う Level 2，行列-行列演算を行う Level 3 が存在する. Level が高いほどデータ再利用性が高く, 高速に計算を行うことが出来る．Level 2 の行列-ベクトル積と Level 3 の行列-行列積には数倍の性能差が存在するため，計算を行列-行列積で構成されるよう変更することで高速化を行うことが可能である．

BLAS の実装には GotoBLAS, ATLAS, ACML, MKL など複数の実装が存在し，今回の実験ではその中の ATLAS を用いて実装を行った．

2.2　固有値計算の一般的手法

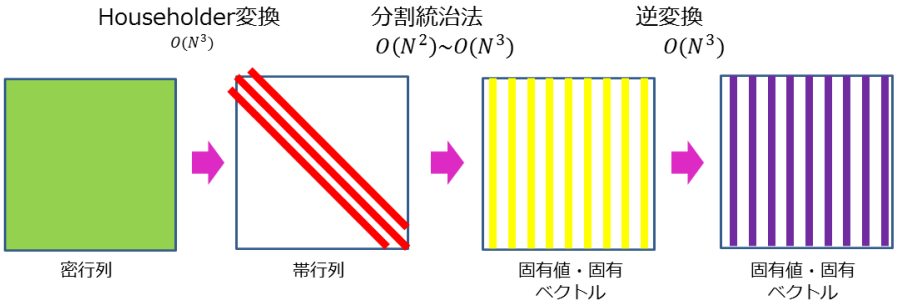

図 1: 固有値計算の一般的手法

実対称行列の固有値問題　式 (2.2.1) は図 1 に示す手順を用いて計算を行うのが一般的である．

$$Ax = \lambda x \qquad (A = A^T) \tag{2.2.1}$$

Householder 法などを用いて密行列を三重対角行列に変換し，三重対角行列に対して QR 法や分割統治法を適用, その後逆変換を行い固有値と固有ベクトルを求める方法である．

この方法を使用することで直接密行列から固有値を計算するよりも高速に計算を行うことができる．三重対角行列への変換は演算量 $O(N^3)$ であり，分割統治法の演算量は $O(N^2) \sim O(N^3)$, 逆変換の演算量は $O(N^3)$ である．分割統治法は他二つの処理と比べて演算量が少なく，行列-行列積を用いて計算を行うことが可能であるため高速に計算できる．また並列化も容易である．逆変換は演算量が $O(N^3)$ と三重対角化操作と同様に大きいが，行列-行列積を中心として構成されるためプロセッサの性能を引き出すことが可能である [3]．それに対して三重対角化操作は計算量が $O(N^3)$ と大きく，単純な方法では行列-ベクトル積が中心となるためプロセッサの性能を引き出すことが難しい．したがって，三重対角化操作が固有値計算全体のボトルネックとなる可能性が高い．このことから本論文では三重対角化操作を高速化する手法を探索する．

第 3 章　Householder 法による三重対角化

Householder 法は実対称行列に対して, 相似変換である Householder 変換を繰り返し作用させることでサイズ $N \times N$ の実対称行列の三重対角化を行うアルゴリズムである．この章では Householder 変換を $N-2$ 回繰り返し作用させる単純な Householder 法のアルゴリズムと, 変換行列をまとめて作用させることでデータ再利用性を向上させた Dongarra らのアルゴリズムについて記述する．次の章で今回の実験に使用した三重対角化操作を二段階に分けることで拡張した Bischof らのアルゴリズムについて記述する．

3.1　Householder 法

サイズ $N \times N$ の実対称行列に対する Householder 法のアルゴリズムは Algorithm 1 である [4].

アルゴリズム中の太字大文字英数字 ($\boldsymbol{A}$) は行列，太字小文字英数字 ($\boldsymbol{a}, \boldsymbol{s}$) はベクトル，大文字と小文字の英数字 ($n, \beta$) はスカラを表す．行列，ベクトルの添字はそれぞれの要素を表している．$\boldsymbol{A}_{[i:m,j:n]}$ は行列 $\boldsymbol{A}$ の i 行から m 行，j 列から n 列までの部分行列を表している．$\boldsymbol{A}_{[i:m,k]}$ は行列 $\boldsymbol{A}$ の i 行から m 行，k 列から構成されるベクトルを表している．$\boldsymbol{A}_{[i,j]}$ は行列 $\boldsymbol{A}$ の i 行 j 列の要素を表している．本論文の以降のアルゴリズム，数式では同様の表記を使用する．

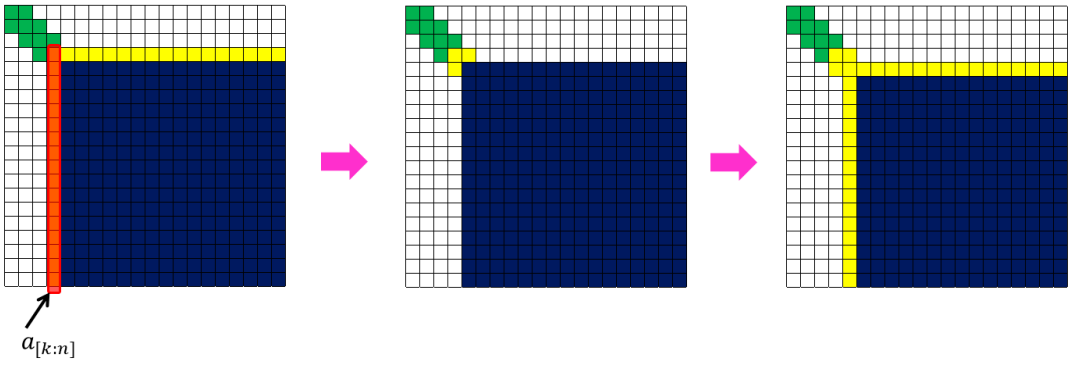

図 2: Householder 法の計算過程

Householder 法は図 2 のように一度の変換で一行，一列の変換を行う相似変換である Householder 変換を行列に対して $N-2$ 回作用させることで左上から右下まで徐々に三重対角化を行う．Householder 変換行列 $\boldsymbol{M}$ は，鏡像変換行列であり式 (3.1.1) ように作成す

Algorithm 1 Householder 法のアルゴリズム

for $k = 1$ to $n-2$ step 1 **do**

[鏡像変換ベクトル u の作成]

$\boldsymbol{a}_{[k:n]} \leftarrow \boldsymbol{A}_{[k:n,k]}$

$s \leftarrow \mathrm{sign}(\boldsymbol{a}_{[k+1]})\|\boldsymbol{a}_{[k:n]}\|$

$$\boldsymbol{a}_{[k:n]} \leftarrow \begin{bmatrix} 0 \\ a_{k+1,k} \\ a_{k+2,k} \\ a_{k+3,k} \\ \vdots \\ a_{n,k} \end{bmatrix} \quad \boldsymbol{b}_{[k:n]} \leftarrow \begin{bmatrix} 0 \\ s \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$\boldsymbol{u} \leftarrow \boldsymbol{a}_{[k:n]} - \boldsymbol{b}_{[k:n]}$

$\beta \leftarrow \frac{2}{\|\boldsymbol{u}\|^2}$

$\boldsymbol{v} \leftarrow \beta \boldsymbol{A}_{[k:n,k:n]}\boldsymbol{u}$

$\alpha \leftarrow \beta \boldsymbol{u}^T\boldsymbol{v}/2$

$\boldsymbol{w} \leftarrow \boldsymbol{v} - \alpha\boldsymbol{u}$

[rank-2 更新]

$\boldsymbol{A}_{[k:n,k:n]} \leftarrow \boldsymbol{A}_{[k:n,k:n]} - \boldsymbol{u}\boldsymbol{w}^T - \boldsymbol{w}\boldsymbol{u}^T$

end for

る．この行列は直交行列かつ対称行列であるため自身が逆行列となり，$\boldsymbol{M} = \boldsymbol{M}^{-1}$ である．

$$\boldsymbol{M} = \boldsymbol{I} - \beta\boldsymbol{u}\boldsymbol{u}^T, \ \beta = \frac{2}{\|\boldsymbol{u}\|} \tag{3.1.1}$$

この時使用するベクトル $\boldsymbol{u}$ は式 (3.1.2) で作成する．この式は第 k 列，k 行の変換を行う際の式である．式において $\|\boldsymbol{u}\|$ でベクトルを除算しているのはノルムを 1 に調整するためである．この式 (3.1.2) では, 鏡像変換ベクトルの作成手順を示すため各ベクトル $\boldsymbol{u}$ について $\|\boldsymbol{u}\|$ を除算しているが，実際のアルゴリズムでは式 (3.1.1) のように $\boldsymbol{u}\boldsymbol{u}^T$ に対してまとめて $\|\boldsymbol{u}\|^2$ で除算を行っている．

$$\begin{aligned} s &= \sqrt{\boldsymbol{A}_{[k+1,k]}^2 + \boldsymbol{A}_{[k+2,k]}^2 + \cdots + \boldsymbol{A}_{[n,k]}^2} \\ &= \|\boldsymbol{A}_{[k+1:n,k]}\| \end{aligned}$$

$$\boldsymbol{a}_{[k:n]} = \begin{bmatrix} 0 \\ a_{k+1,k} \\ a_{k+2,k} \\ \vdots \\ a_{n,k} \end{bmatrix}, \quad \boldsymbol{b}_{[k:n]} = \begin{bmatrix} 0 \\ s \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.1.2}$$

$$\begin{aligned} \boldsymbol{u} &= \boldsymbol{a}_{[k:n]} - \boldsymbol{b}_{[k:n]} \\ \boldsymbol{c} &= \frac{1}{\|\boldsymbol{u}\|} \\ \boldsymbol{v} &= \boldsymbol{c}\boldsymbol{u} \end{aligned}$$

Householder 変換において，$\boldsymbol{A} = \boldsymbol{M}\boldsymbol{A}\boldsymbol{M}$ を単純に計算した場合演算量は約 $2N^3$ である．$\boldsymbol{M}\boldsymbol{A}\boldsymbol{M}$ を $\boldsymbol{M} = \boldsymbol{I} - \beta\boldsymbol{u}\boldsymbol{u}^T$ として分解し，次の式 (3.1.3) のように整理することで，rank-2 更新となり演算量を約 N^2 に削減することが出来る．

$$\begin{aligned} \boldsymbol{M}\boldsymbol{A}\boldsymbol{M} &= \left(\boldsymbol{I} - \beta\boldsymbol{u}\boldsymbol{u}^T\right)\boldsymbol{A}\left(\boldsymbol{I} - \beta\boldsymbol{u}\boldsymbol{u}^T\right) \\ &= \boldsymbol{A} - \beta\boldsymbol{A}\boldsymbol{u}\boldsymbol{u}^T - \beta\boldsymbol{u}\boldsymbol{u}^T\boldsymbol{A} + \beta^2\boldsymbol{u}\boldsymbol{u}^T\boldsymbol{A}\boldsymbol{u}\boldsymbol{u}^T \\ &= \boldsymbol{A} - \beta\left(\boldsymbol{A}\boldsymbol{u}\right)\boldsymbol{u}^T - \beta\boldsymbol{u}\left(\boldsymbol{A}\boldsymbol{u}\right)^T + \beta^2\alpha\boldsymbol{u}\boldsymbol{u}^T, \quad \alpha = \boldsymbol{u}^T\left(\boldsymbol{A}\boldsymbol{u}\right) \\ &= \boldsymbol{A} - \beta\boldsymbol{u}\left(\left(\boldsymbol{A}\boldsymbol{u}\right)^T - \alpha\boldsymbol{u}^T\right) - \beta\left(\left(\boldsymbol{A}\boldsymbol{u}\right) - \alpha\boldsymbol{u}\right)\boldsymbol{u}^T, \quad \alpha = \boldsymbol{u}^T\left(\boldsymbol{A}\boldsymbol{u}\right) \\ &= \boldsymbol{A} - \boldsymbol{u}\boldsymbol{w}^T - \boldsymbol{w}\boldsymbol{u}^T, \quad \alpha = \boldsymbol{u}^T\left(\boldsymbol{A}\boldsymbol{u}\right), \quad \boldsymbol{w} = \beta\left(\boldsymbol{A}\boldsymbol{u} - \alpha\boldsymbol{u}\right) \end{aligned} \tag{3.1.3}$$

Householder 法のアルゴリズムは，行列-ベクトル積と行列の rank-2 更新がそれぞれ約 $(2/3)N^3$ の演算量であり，それぞれが全演算量 $(4/3)N^3$ の約半分である．行列サイズ N の行列-ベクトル積と rank-2 更新では演算回約数 N^2 とデータがほぼ一対一対応にあり，データ再利用性が悪いという問題点が存在する．データ再利用性が低い場合，プロセッサ性能よりも大幅に小さいメモリバンド幅によって性能が制限されてしまうため非常に低速である [3].

3.2 Dongarra らのアルゴリズム

Dongarra らは Householder 法の低いデータ再利用性を解消するためアルゴリズム中の rank-2 更新を rank-2K 更新へ変更した [5, 6]. Algorithm 2 のように行列右下部分に対する rank-2 更新を各反復で行うのではなく，K 段の反復に一回 K 段分の rank-2 更新をまとめて行う rank-2K 更新へ変更した．

図 3 に示す計算過程において $l \sim l + K$ 段の Householder 変換のためのに必要な鏡像変換ベクトル $\boldsymbol{u}^{(l\sim l+K)}$ は赤い線で囲まれた部分から計算される．Householder 法の rank-2 更新では各反復で行列の右下部分 $A_{l:n,l:n}$ を全て更新していたが，実際に次の反復の変換行列作成に使用する部分は赤い線で囲まれた部分のみである．そのため (1) で必要な部分のみの

Algorithm 2 Dongarra らのアルゴリズム

$\boldsymbol{U} = O$, $\boldsymbol{C} = O$

for $l = 1$ to $n - 2$ step K **do**

for $p = 0$ to $K - 1$ step 1 **do**

$k = l + p$

[部分 Householder 変換，次の変換行列作成に必要な部分のみ更新]

(1) $\boldsymbol{a}^{(k)}_{[k:n]} \leftarrow \boldsymbol{A}_{[k:n,k]} - \boldsymbol{U}_{[l:n,l:k]}(\boldsymbol{C}_{[l:n,l:k]})^T - \boldsymbol{C}_{[l:n,l:k]}(\boldsymbol{U}_{[l:n,l:k]})^T$

[鏡像変換ベクトル u の作成]

$s \leftarrow sign(\boldsymbol{a}^{(k)}_{[k+1]})\|\boldsymbol{a}^{(k)}_{[k+1:n]}\|$

(2) $\boldsymbol{u^{(k)}} \leftarrow [\boldsymbol{a}^{(k)}_k, \boldsymbol{a}^{(k)}_{k+1} + s, \ldots, \boldsymbol{a}^{(k)}_n]$

(3) $\beta \leftarrow \frac{2}{\|\boldsymbol{u}^{(k)}\|}$

[行列，ベクトル積]

(4) $\boldsymbol{p} \leftarrow \beta(\boldsymbol{A}_{[k:n,k:n]} - \boldsymbol{U}_{[l:n,l:k]}\boldsymbol{C}^T_{[l:n,l:k]} - \boldsymbol{C}_{[l:n,l:k]}\boldsymbol{U}^T_{[l:n,l:k]})u^{(k)}$

(5) $\boldsymbol{v} \leftarrow \beta \boldsymbol{u}^{(k)T}\boldsymbol{p}/2$

(6) $\boldsymbol{c}^{(k)} \leftarrow \boldsymbol{p} - \boldsymbol{v}\boldsymbol{u^{(k)}}$

$\boldsymbol{U}_{[l:n,k+1]} \leftarrow \boldsymbol{u}^{(k)}$

$\boldsymbol{C}_{[l:n,k+1]} \leftarrow \boldsymbol{c}^{(k)}$

end for

[rank-2K 更新]

$q = l + K$

(7) $\boldsymbol{A}_{[q:n,q:n]} \leftarrow \boldsymbol{A}_{[q:n,q:n]} - \boldsymbol{U}_{[l:n,l:q]}\boldsymbol{C}^T_{[l:n,l:q]} - \boldsymbol{C}_{[l:n,l:q]}\boldsymbol{U}^T_{[l:n,l:q]}$

end for

計算を行い, (4) で rank 更新の補正計算を行う．(7) で K 回の反復ごとに一回更新されなかった行列の残りの右下部分 $\boldsymbol{A}_{[l+K:n,l+K:n]}$ に対してまとめて rank-2K 更新を行う．

rank-2 更新の代わりに rank-2K 更新を使用することで，行列-ベクトル積をデータ再利用性が高い行列-行列積に置き換えることが出来る．そのため Householder 法と比較して実行時間を約 1/2 程度に削減することが出来る [3]．しかし，rank-2K 更新を除いた他の演算の多くは依然としてデータ再利用性が低い行列-ベクトル演算により構成され，性能向上のボトルネックとなる問題が存在する．

LAPACK における実対称行列の三重対角化関数 SYTRD ではこの方式を改良したものが使用されている [7].

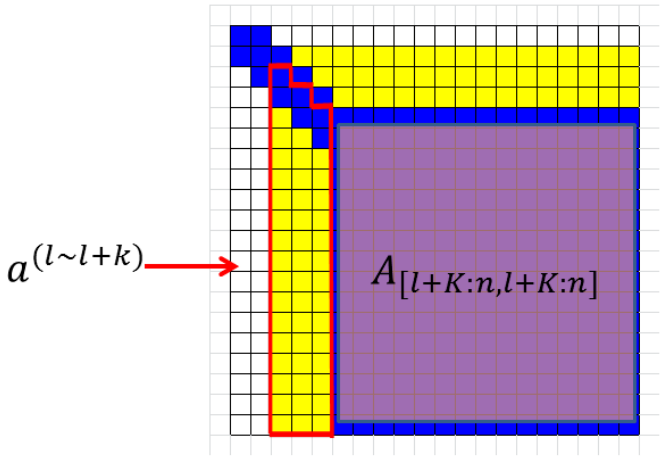

図 3: Dongarra らのアルゴリズムの計算途中の行列

第 4 章　Bischof 法による三重対角化

Bischof らは，密行列を三重対角行列に直接変換するのではなく，図 4 のように帯行列化を行い一度帯行列へ変換した後，帯幅縮小操作を行うことで多段階に三重対角行列へ変換する，多段階三重対角化の方法 (以下 Bischof 法) を提案した [8]. 帯行列化については WY Representation[9] を用いて行う方法や，村上らによるブロック鏡像変換を使用する方法 [10] が存在する．帯幅縮小操作には村田法 [11] や Rutishauser 法 [13] と呼ばれる方法が存在する．

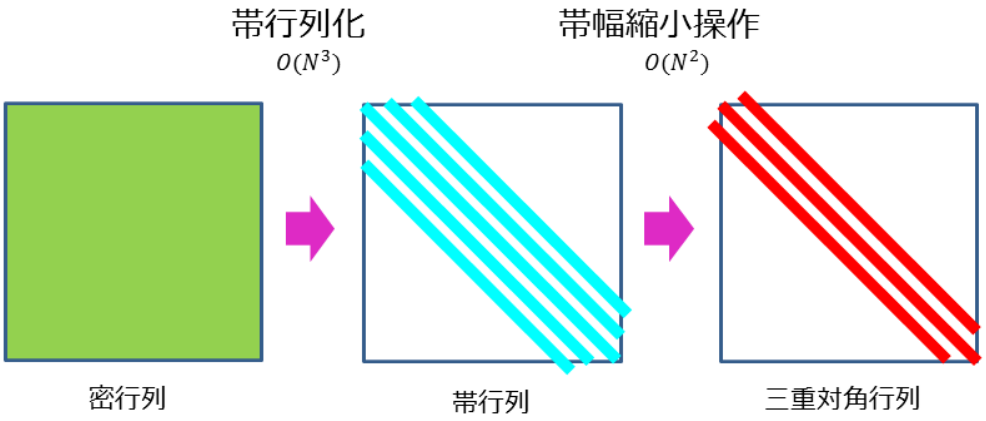

図 4: Bischof らによる二段階三重対角化手順

本章では本論文の実験に使用する村田法による帯幅縮小操作について詳しく記述する．

4.1　村田法

村田らは帯幅縮小操作において，通常の Householder 変換による帯幅縮小を行った場合帯外に非ゼロ要素が生成され，密行列に戻ってしまう問題を回避する方法 (以下, 村田法) を提案した [11].

$$\begin{bmatrix} \boldsymbol{I} & 0 & 0 \\ 0 & \boldsymbol{M} & 0 \\ 0 & 0 & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{E} & \boldsymbol{B}^T & \boldsymbol{F}^T \\ \boldsymbol{B} & \boldsymbol{A} & \boldsymbol{C}^T \\ \boldsymbol{F} & \boldsymbol{C} & \boldsymbol{G} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} & 0 & 0 \\ 0 & \boldsymbol{M} & 0 \\ 0 & 0 & \boldsymbol{I} \end{bmatrix} = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{B}^T\boldsymbol{M} & \boldsymbol{F}^T \\ \boldsymbol{M}\boldsymbol{B} & \boldsymbol{M}\boldsymbol{A}\boldsymbol{M} & \boldsymbol{M}\boldsymbol{C}^T \\ \boldsymbol{F} & \boldsymbol{C}\boldsymbol{M} & \boldsymbol{G} \end{bmatrix} \tag{4.1.1}$$

式 (4.1.1) のように, 行列の一部分 $\boldsymbol{A}$ に対して対象部分を変換する行列 $\boldsymbol{M}$ を左右から作用させた場合，$\boldsymbol{A}$ に隣接した上下左右の部分行列に対しても行列が作用してしまう．このことから帯幅縮小の際に半帯幅 nb の帯行列の一部分に対して通常の Householder 変換を用

Algorithm 3 村田法のアルゴリズム

for $k = 1$ to $n-2$ step 1 **do**

[帯幅縮小操作]

$\boldsymbol{a}_{[k:k+nb+1]} \leftarrow \boldsymbol{A}_{[k:nb+1,k]}$

(1) [Householder 変換行列作成操作,$\boldsymbol{u}, \boldsymbol{c}, \beta$ を作成]

[行列の rank-2 更新]

(2) $\boldsymbol{A}_{[k:nb+1,k:nb+1]} - \boldsymbol{u}\boldsymbol{c}^T - \boldsymbol{c}\boldsymbol{u}^T$

if $k + m > n$ **then**

[bulge 作成]

$i1 = k + nb + 1;\ i2 = k + 2nb + 1;\ j1 = k + 1;\ j2 = k + nb + 1;$

$\boldsymbol{p} \leftarrow \beta \boldsymbol{A}_{[i1:i2,j1:j2]}\boldsymbol{u}$

(3) $\boldsymbol{A}_{[i1:i2,j1:j2]} \leftarrow \boldsymbol{A}_{[i1:i2,j1:j2]} - \boldsymbol{p}\boldsymbol{u}^T$

[bulge 追跡操作]

for $l = 1$ to $k + l \times nb + 1 < n$ step nb **do**

$t = k + l \times nb + 1$

(4) [鏡像変換ベクトルの作成]

$\boldsymbol{a}_{[k:n]} \leftarrow \boldsymbol{A}_{[t:t+nb,t-nb]}$

(5)[Householder 変換行列作成操作, $\boldsymbol{u}$, $\boldsymbol{c}$, β を作成]

[Householder 変換行列を左から作用させて bulge を削除]

$i1 = t;\ i2 = t + nb;\ j1 = t - nb;\ j2 = t;$

$\boldsymbol{p} \leftarrow \boldsymbol{A}^T_{[i1:i2,j1:j2]}\boldsymbol{u}$

(6)$\boldsymbol{A}_{[i1:i2,j1:j2]} \leftarrow \boldsymbol{A}_{[i1:i2,j1:j2]} - \beta \boldsymbol{u}^T \boldsymbol{p}$

[rank-2 更新により Householder 変換を対角部分に左右から作用させる]

(7)$\boldsymbol{A}_{[t:t+nb,t:t+nb]} \leftarrow \boldsymbol{A}_{[t:t+nb,t:t+nb]} - \boldsymbol{u}\boldsymbol{c}^T - \boldsymbol{c}\boldsymbol{u}^T$

if $k + l \times nb + nb < n - 1$ **then**

[新たな bulge 作成]

$i1 = t + nb + 1; i2 = t + 2nb + 1; j1 = t; j2 = t + nb;$

$\boldsymbol{p} \leftarrow \boldsymbol{A}_{[i1:i2,j1:j2]}\boldsymbol{u}$

(8)$\boldsymbol{A}_{[i1:i2,j1:j2]} \leftarrow \boldsymbol{A}_{[i1:i2,j1:j2]} - \beta \boldsymbol{p}\boldsymbol{u}^T$

end if

end for

end if

end for

いて帯幅縮小を行った場合, 図 5 のように隣接する帯外の部分に対しても変換行列が作用し，bulge と呼ばれる非ゼロ要素が生成される．削除する帯幅が拡大したため，次の変換行列のサイズが増大し，更に大きな非ゼロ要素である bulge が帯外に生成される．発生したより大きな bulge によって，また次に削除する帯幅が拡大したため変換行列のサイズが拡大し，更に大きな非ゼロ要素である bulge が帯外に生成される．この操作を繰り返すことにより，帯幅が徐々に拡大し，帯行列が密行列に変化してしまう．そのため帯行列に直して計算を行ったのにも関わらず，元の密行列に対して計算を行った場合とほぼ同様の演算回数とメモリ容量を必要とすることになってしまう．

帯幅縮小操作によって，帯行列が密行列に戻ることを避けるために，村田法では Algorithm 3 [12] のようにすることで，帯幅の拡大を抑えながら帯幅縮小を行う．rank-2 更新による Householder 変換を用いた帯幅縮小の際に発生する帯外の非ゼロ要素である bulge を削除するため図 6 のような操作を行う．図 6 の操作では発生した bulge の削除する列から鏡像変換ベクトルと，そのベクトルを使用した Householder 変換行列を作成する．この変換行列を bulge に対して片側から作用させることで bulge の削除を行う．変換を行う対象となる行列 $\boldsymbol{A}$ は対称行列なので，片側 bulge の対称部分にも変換行列を作用させる必要がある．式 (4.1.1) から行列の対角部分に左右から変換行列を作用させることで，bulge 部分である $\boldsymbol{B}$ とその対称 bulge $\boldsymbol{B}^T$ に対して変換行列を片側から作用させることが出来ることがわかる．これらのことから bulge から Householder 変換行列を作成し，対角部分に左右から作用させることで bulge 削除操作を行うことが可能である．bulge 削除操作の結果新たに帯外に bulge が発生するため，発生した bulge に対して同様に削除を行う．この bulge 削除操作を行列の右下部分まで繰り返す．行列の右下部分では図 7 のように，変換行列の作用に伴い発生する bulge が行列の外に出るため bulge は発生しない．

bulge 削除操作を行列の右下部分で bulge が発生しなくなるまで繰り返す一連の操作を bulge 追跡と呼ぶ. bulge 追跡における計算範囲をずらしながら徐々に変換を行う様子を示したものが図 8 である．図 8 の色がついている部分が非ゼロ要素であり，黄色，緑色，桃色の部分が各反復で変換を行う部分である．黄色が帯幅縮小操作と bulge 消去を行う部分，緑色が変換に伴い発生した bulge 部分，桃色が bulge 消去の際中心となる対角部分である．途中左下の図で複数の反復をまとめて表現している部分があるが，反復が進むにつれて計算範囲が左上から右下までずれていき，右下まで達した後再び左上に戻って繰り返していることを表している．

一度の bulge 追跡では複数列存在する bulge のうち，一列しか削除を行うことが出来ないが，次の帯幅縮小で必要な部分のみ削除すれば良いため，一度の bulge 追跡で十分である．bulge 追跡を行い，bulge を削除した後に次の帯幅縮小操作を行うと，また新たに bulge が発生する．発生した bulge に対して bulge 追跡を行い，次の帯幅縮小操作を行う．この操作を繰り返すことで帯行列を三重対角行列に変換する．

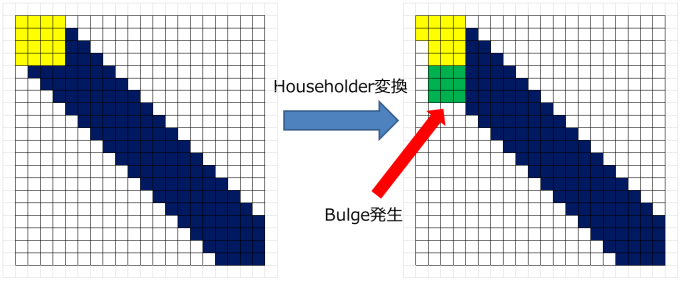

図 5: Householder 変換による帯幅縮小に伴う bulge の発生

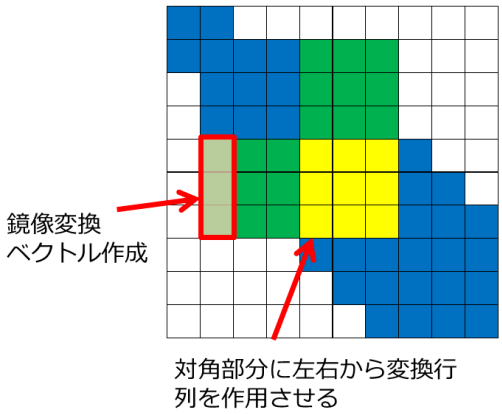

図 6: bulge 削除操作

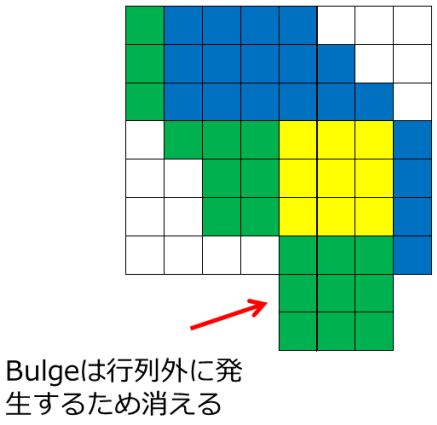

図 7: 行列右下部分に対する変換による bulge 追跡終了

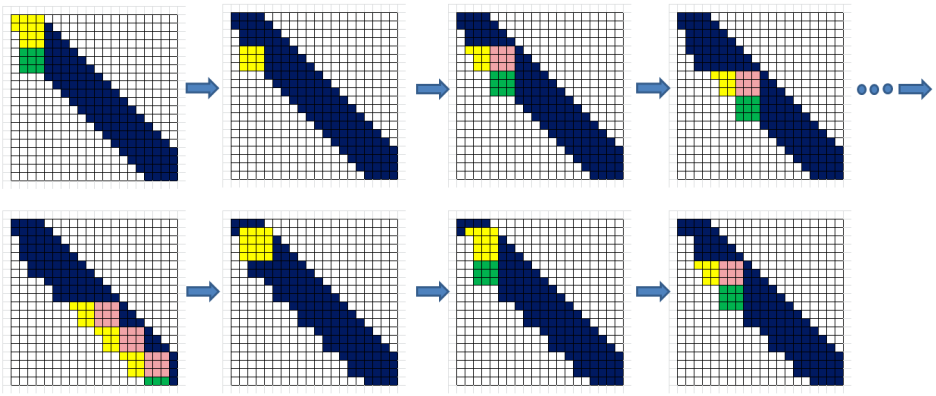

図 8: 村田法による bulge 追跡過程

村田法による帯幅縮小操作の演算回数は約 $6nb \times N^2$ 回であり，メモリ量も半帯幅の二倍 $2nb$ で計算することが可能になる [12]．帯行列に対して Householder 法を用いた場合の演算量が約 $2N^3/3$ であり，メモリ量が約 N^2 であるのに対して演算量，メモリ量ともに優れおり，データ局所性も向上しているためキャッシュ効率も良くなる．また村田法の各ステップにおける行列, ベクトルに対する演算の各領域は連続して存在しているため BLAS などの各環境に最適化された高速線形演算ライブラリを使用することが容易であり，高速に計算を行うことが出来る．

4.2 Rutishauser 法

Rutishauser は特定の帯幅の対角帯行列を三重対角帯行列に縮小する方法 (以下, Rutishauser 法) を提案した [13]．Schwarz は Rutishauser のアルゴリズムを元に，任意の帯幅に適用できるよう拡大した [14]．Kaufman は Schwarz のアルゴリズムを元に改良を行い，一度の変換でより広い範囲を同時に変換できるようにした [15]．LAPACK による帯行列の三重対角化 SBTRD は Kaufman のアルゴリズムを元にしたものが使用されている．

行列サイズ $N \times N$ の実対称行列 A は式 (4.2.1) のように行列 A に左右から Givens 回転行列 Q を作用させることで三重対角行列 T に変換を行うことが出来る．

$$QAQ^T = T \tag{4.2.1}$$

Q は約 $(nb-1) \times N^2/(2nb)$ 回の平面 Givens 回転から構成されている．村田法では Householder 法を用いて左上から右下まで要素の削除と帯外に発生する要素の追跡を行ったが，Rutishauser 法では Givens 回転を使用して徐々に削除を行う．

Rutishauser 法による帯幅縮小操作は Algorithm 4 のように表現される．1:で半帯幅分だけループを行う．一度のループで一行の縮小を行う．2:で行列の左上から右下まで反復を行

Algorithm 4 Rutishauser 法のアルゴリズム

```
1: for i = nb to 1 < i step −1 do
2:    for k = 1 to N − 2 step 1 do
3:       [Givens 回転による帯幅縮小操作]
4:        Givens(i,k)
5:       [帯外要素追跡操作]
6:       for l = 0 to k + l ∗ i + i − 1 < n step 1 do
7:          [Givens 回転による帯外要素削除]
8:           Givens(i,k+l*i+i-10)
9:       end for
10:   end for
11: end for
```

う．4:で Givens 回転を用いて一箇所の帯幅縮小操作を行う．6:で 4:の Givens 回転によって生じた帯外要素を右下まで反復して追いかける. 8:で Givens 回転により帯外に発生した余分な要素を削除する．

このアルゴリズムは Schwarz[14] らによる最も単純な方法で，一度の Givens 回転による要素の削除と追跡により，一つの要素しか削除を行うことが出来ない. 帯幅縮小操作と，それに伴う追跡処理を $N-2$ 回作用させることで帯幅を 1 縮小し，三重対角化を行うためには半帯幅 nb になるまで帯幅を 1 ずつ縮小していく．そのため，行列の端から端までの局所性の低いアクセスと計算を多数回行うことになり非常に低速になる．Kaufman [15] らは一度に複数の Givens 回転を行うことで改良を行った.

Rutishauser 法は平方根計算が多く, 局所性もあまり高くない．計算順序を変更することで BLAS の使用が可能になり，高速化を行うことが出来るようになるが，やはり BLAS Level が高くないため性能を完全に引き出すことが難しいという問題が存在する.

第 5 章　村田法の並列化

村田法と Rutishauser 法を比較した場合，村田法の方が行列-ベクトル積を中心に構成されておりデータ再利用性が高い．また村田法で使用する変換行列は長方形など単純な形であり，BLAS を適用できる箇所が多く，圧縮形式に直した場合もほぼ BLAS を使って計算が可能である．そのため各環境に合わせてソースコードを書き直すことが少なくなる．村田法は帯幅が一定以上大きい場合高速であり，Rutishauser 法は帯幅が小さい場合高速である．帯行列化と合わせて考えた時，高速に計算が可能な帯幅は少なくとも 50 は超える．以上のことより本論文では Householder 変換を使用した村田法に注目し，さらなる高速化の検討を行った．

村田法は図 8 のように，一度の反復で左上から右下まで計算範囲をずらしながら反復計算を行う．bulge 追跡操作が一定以上進んだ場合，図 9 のように bulge 追跡操作の計算範囲と, 次の反復における帯幅縮小操作と bulge 追跡に使用する計算範囲が重複していないため並列化による同時実行が可能である．bulge 追跡 (1) が一定以上進み，次の帯幅縮小の計算範囲と重ならなくなった時点で次の帯幅縮小 (2) を先行した bulge 追跡 (1) と同時に実行する．bulge 追跡 (1) と帯幅縮小 (2) に伴う bulge 追跡 (3) が一定以上進み，また計算範囲が重ならなくなった時点で次の帯幅縮小 (4) を先行した bulge 追跡 (1), (3) と同時に実行する．bulge 追跡 (1), (3) と帯幅縮小 (4) に伴う bulge 追跡 (5) が一定以上進み，また計算範囲が重ならなくなった時点で次の帯幅縮小を行う．この操作を繰り返すことで村田法の並列実行が可能になる．この方法を用いて村田法の並列化を行った．

各反復では行列の左上から右下まで移動しながら帯行列の全ての要素を使用し，次の反復における計算は，前の反復の計算結果を必要とする．したがって頻繁に各プロセス，スレッド間で行列データの同期を取る必要がある．MPI などを用いたプロセス並列化を行った場合，頻繁にコストが大きいプロセス間通信を伴う同期が発生し低速化する可能性がある．そのため同期コストの小さいスレッド並列化を行った．

一回の帯幅縮小操作とそれに伴う bulge 追跡操作を一つのタスクとし，各スレッドはタスクをそれぞれ実行する．bulge 追跡の途中，計算範囲がずれることで次の帯幅縮小が実行可能になると，次スレッドが帯幅縮小と bulge 追跡を開始する．各スレッドは先行スレッドの計算範囲と重ならないように共有メモリ領域に存在するフラグを用いた重複防止処理を行っている．重複防止処理はスレッド間で排他的処理を行うため一定のコストが発生する．そのため重複防止方式によって同期処理や排他的処理の回数が変化する場合，実行速度に影響を及ぼすことが考えられる．また重複防止処理によって次スレッドの実行開始を判断するため，並列性に影響を与えることも考えられる．今回複数の重複防止方式を実装し性能の比較を行う．

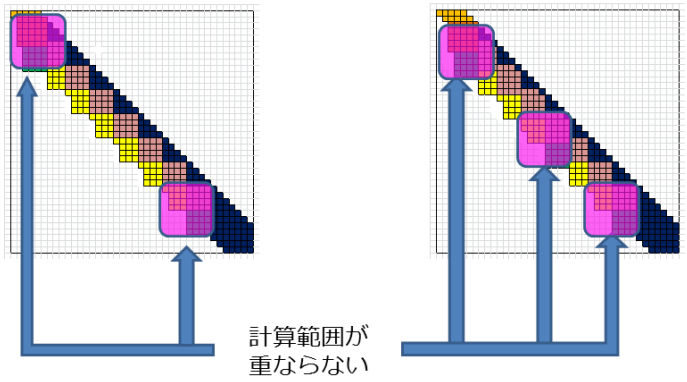

図 9: 村田法の並列化

5.1 重複防止方式

村田法の並列化を行う場合計算範囲の重複や，先行スレッドの追い越しを防ぐために頻繁に重複防止処理を行う必要がある．重複防止処理は全スレッド間でメモリの内容を一致させる同期処理や，共有メモリ領域へのアクセスなどである．重複防止処理はスレッド間で排他的処理やデータ通信を行うため一定のコストが発生する．そのため重複防止方式によって同期回数や排他的アクセス回数が変化する場合，実行時間に影響を及ぼすことが考えられる．

村田法における共有メモリ領域に存在するフラグを用いた重複防止方式には複数の方法が考えられる．今回その中の 3 つの方法を実装し性能評価を行った．3 つの形式それぞれ静的方式を使用した．静的方式各スレッドがどのタスクを実行するか事前に規定されており，各スレッドが決められた順番通りにタスクを実行する方式である．

5.2 閉塞方式による重複防止

閉塞方式は 1 つの区間に一つのスレッドのみが存在し，他のスレッドは区間が空くまで待機する方式である．今回の村田法に対して閉塞を適用するため，図 10 のように行列をいくつかの区間に分割し，各区間に対応するフラグを共有メモリ領域に作成した．このフラグを確認することで，対応する区間にスレッドが存在するか判別を行う．他スレッドがフラグを確保しておらず，スレッドが対応区間に存在していない場合フラグを確保し，対応区間に対する帯幅縮小や bulge 削除処理を行い，次の領域に対するフラグの確認を行う．閉塞による処理を簡単に表したものが図 11 である．それぞれのセルは区間を表し，色がついたセルはスレッドがその区間に存在することを表す．計算の際にはスレッドは左から右に区間をずらしながら進んでいく．1.~13. は時間の経過を表し，左から右にスレッドが

区間をずらしながら進んでいること，複数のスレッドが同時に計算を行っていることを示している．

1. ではスレッドは存在しておらず，2. でスレッド 1 が区間を確保する．2. 3. 4. においてスレッド 1 は徐々に進み，区間に空きが出来た際にスレッド 2 がフラグを確保して区間に入る．5.~8. でスレッド 1, 2 は区間をずらしながら進み，9. で空いた区間にスレッド 3 が入る．11. で区間の端まで達したスレッド 1 は 13. で左からまた区間を確保して進んでいる．この操作を繰り返すことで帯幅縮小と bulge 追跡操作において計算範囲の重複と先行スレッドの追い越しを防ぐ．

共有領域に存在するフラグに各スレッドがアクセスを行うため，フラグに対して同時にアクセスを行い一貫性が崩れる可能性がある．そのためフラグに対するアクセスは排他的処理を行う．また後続スレッドによる追い越しを避けるため図 11 のようにフラグを確保し次の区間に進む際，先に次の区間のフラグを確保した後に，進んだ分の区間を開放する．この操作によって後続スレッドが先行スレッドのフラグと計算位置が重なることを防ぐ．

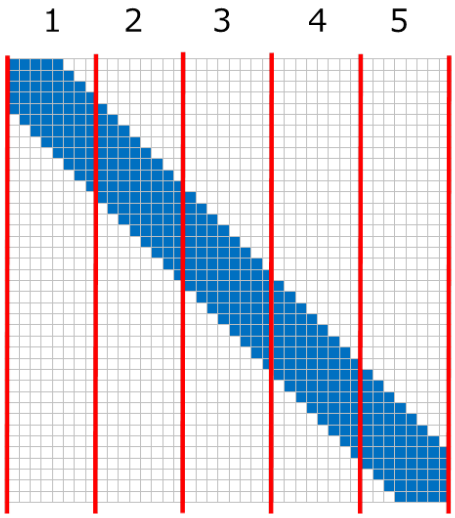

図 10: 行列の区間分割

今回の実験で使用した動的な閉塞方式による重複防止処理を行い並列に計算を行うアルゴリズムは Algorithm 5 である．

次の帯幅縮小を実行する位置を表す i と各区間にスレッドが存在するかを表すフラグは全スレッド間で共有を行い，アクセスする際には排他処理を行う．

1:で全ての帯幅縮小が完了するまでループを行う．2:で i 番目の帯幅縮小が自スレッドに割り当てられている場合，使用する区間に他のスレッドが入っていないかフラグを確認する．すでに他スレッドが存在している場合 1:まで戻りループを続けることでフラグが開放されるまでビジーウェイトを行う．他のスレッドが次に区間に存在しない場合 3:の Get_Flag(i) で i 番目の帯幅縮小に使用する区間のフラグを確保する．フラグを確保出来た

図 11: 閉塞方式によるフラグ確保の模式図

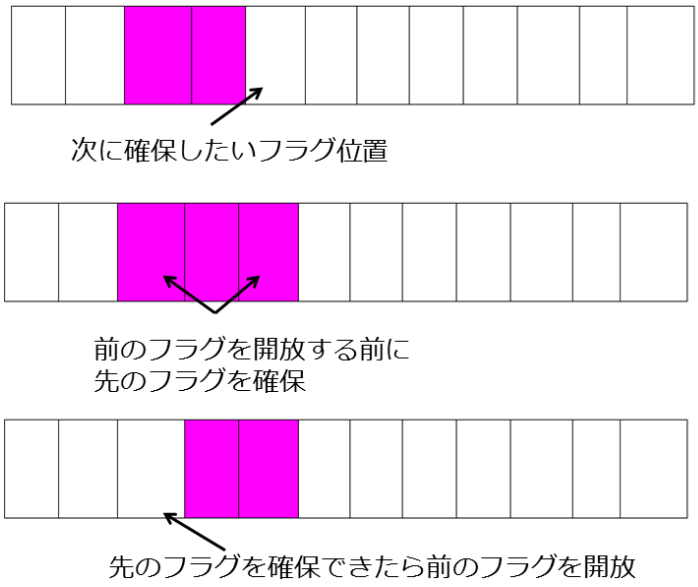

図 12: 閉塞方式において進む際の処理の模式図

Algorithm 5 閉塞方式による重複防止アルゴリズム

Parallel: { スレッド数分並列に実行する }

```
1:  while i < N − 2 do
2:    if Check_Flag(i) then
3:      Get_Flag(i)
4:      Band_Reduction(i)
5:      k = i + 1
6:      i = i + 1
7:      l = 1
8:      while k + l × nb + 1 < n do
9:        if Check_Flag(l) then
10:         Get_Flag(l)
11:         Bulge_Erace(l)
12:         l = l + 1
13:       end if
14:     end while
15:     Release_Flag()
16:   end if
17: end while
```

場合 4:の Band_Reduction(i) で i 番目の帯幅縮小を実行し，i を 1 増やし次の帯幅縮小の位置を指定する．8:で bulge 追跡が完了するまでループを行う．9:の Check_flag(l) で i 番目の帯幅縮小に伴う l 回目の bulge 削除に使用する区間に他のスレッドが存在しているか確認する．他のスレッドが存在している場合 8:に戻りループを続けることでフラグが開放されるまでビジーウェイトを行う．他スレッドが次の区間に存在しない場合，10:の Get_Flag(l) で l 回目の bulge 削除に使用する区間のフラグを確保し，前の区間を開放する．フラグを確保できた場合 11:の Bulge_Erace(l) で l 回目の bulge 削除を実行し，l を更新して次の bulge 追跡の位置を示す．bulge 追跡が完了した時，15:の Release_Flag で追跡に使用した区間に対応するフラグの開放を行う．

5.3 メッセージ方式による重複防止

メッセージ方式はスレッド間でフラグを更新することにより実行完了した位置を知らせるメッセージを送る方式である．

メッセージ方式による重複防止処理の模式図は図 13 である．黄色と赤の小さな四角は各スレッドに対応したフラグを表し，帯幅縮小や bulge 削除を行う際，各スレッドは対応する自身のフラグを確認する．黄色の時実行不可であり，赤の時実行可能である．各スレッドは青と緑の四角で表されている．青の時は待機中であり，処理を実行していない．緑の時は実行中であり，帯幅縮小や bulge 削除を実行している．図 13 の番号は時間の経過を表し，スレッドは右からタスクを実行していく．

初期状態 1:ではスレッドは処理を実行しておらず右端のスレッド 1 が実行可能になっている．初期状態ではスレッド 1 に先行スレッドが存在しないため帯幅縮小と bulge 削除をまとめて最後まで進行できるフラグが設定されている．2:でフラグを確認したスレッド 1 が処理を実行する．3:でスレッド 1 は処理を実行しながらフラグを更新し，後続スレッドに処理が一部進行したことを通知する．4:でスレッド 1 からフラグの更新を受け，進行可能になったスレッド 2 が処理を実行する．スレッド 1 は処理を実行しながらフラグを更新し，後続スレッドに処理が進行したことを通知する．5:でスレッド 2 はスレッド 1 からのフラグ更新を受け，進行可能であるため処理を実行しながら，フラグを更新することで後続スレッドに処理が進行したことを通知する．スレッド 1 は一回の帯幅縮小とそれにともなう bulge 追跡が完了したため，後続スレッドに処理が完了したことを通知する．6:でスレッド 1 は処理が完了したため，次の帯幅縮小を行うが，フラグが更新されていないため待機する．スレッド 2 はスレッド 1 からのフラグ更新により先行スレッドがすでに処理を完了しており，進行可能のであるため，処理を実行しながら，フラグを更新することで後続スレッドに処理が進行したことを通知する．スレッド 3 はスレッド 2 からのフラグ更新を受け，進行可能であるため処理を実行しながら，フラグを更新することで後続スレッドに処理が進行したことを通知する．7:でスレッド 2 は後続スレッドに一連の処理が

完了したことをフラグ更新により通知する．スレッド 3 はスレッド 2 からのフラグ更新を受け，進行可能であるため処理を実行しながら，後続スレッドに処理が完了したことを通知する．8:, 9:で同様の処理を行う．スレッド 4 の後続スレッドはスレッド 1 になる循環構造になっている．

各スレッドはフラグを確認し進行できない場合，ループを行いフラグを確認し続けビジーウェイトを行っている．

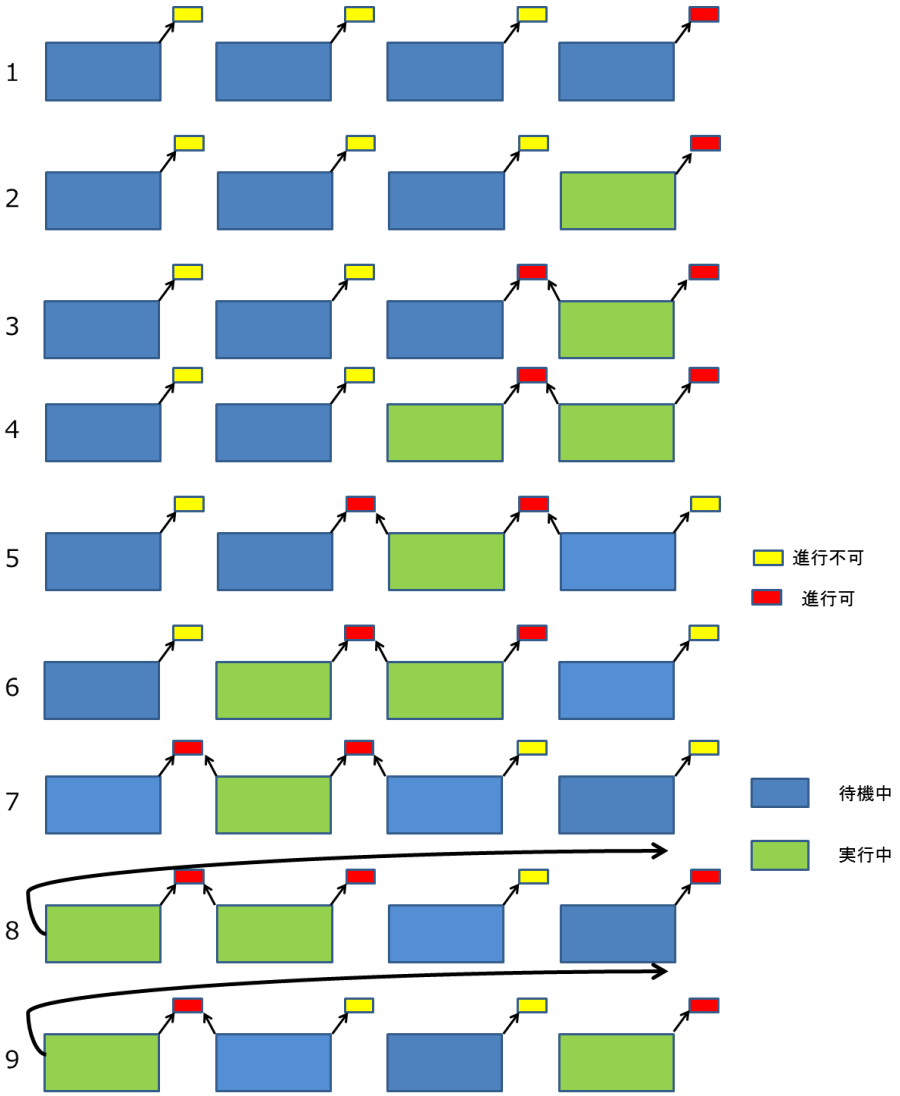

図 13: メッセージ方式の模式図

今回の実験で使用した静的なメッセージ方式による重複防止を行い並列に計算を行うアルゴリズムは Algorithm 6 である．

先行スレッドがどこまで進んだかを表すフラグを全スレッド間で共有し，アクセスする際

Algorithm 6 メッセージ方式の重複防止アルゴリズム

Parallel: { スレッド数分並列に実行する }

```
 1: i = thread_num
 2: while i < n − 2 do
 3:   if Check_Flag(i, thread_id) then
 4:     Band_Reduction(i)
 5:     Send_Message(i, thread_id)
 6:     k = i + 1
 7:     l = 1
 8:     while k + l × nb + 1 < n do
 9:       if Check_Flag(l, thread_id) then
10:         Bulge_Erace(l)
11:         Send_Message(l, thread_id)
12:         l = l + 1
13:       end if
14:     end while
15:     i = i + thread_num
16:   end if
17: end while
```

には排他処理を行う．*thread_id* は各スレッドを表すユニークな ID, *thread_num* はスレッド数である．

2:で割り振られた回数帯幅縮小を実行するまでループを行う．3:の Check_flag(*i*, *thread_id*) で *i* 番目の帯幅縮小が実行出来るか自スレッドに対応するフラグを確認する．フラグ確認の結果先行スレッドが計算範囲が重複しなくなるまで進んでいない場合，2:に戻りループを続け，ビジーウェイトを行う．フラグを確認し，先行スレッドが次の帯幅縮小と計算範囲が重複しなくなるまで進んでいる場合，4:の Band_Reduction(*i*) で *i* 番目の帯幅縮小を実行する．帯幅縮小が完了した後, 5:の Send_Message(*i*, *thread_id*) で自スレッドに対応するフラグの更新を行い，進行したことを通知する．8:で bulge 追跡が完了するまでループを行う．9:の Check_Flag(*l*, *thread_id*) で *l* 回目の bulge 削除が実行可能か自スレッドに対応するフラグを確認する．先行スレッドが十分進行していない場合，8:に戻りループを続け, 先行スレッドが進行しフラグが更新されるまでビジーウェイトを行う．フラグを確認し，先行スレッドが次の *l* 回目の bulge 削除と重複しなくなるまで進行している場合，10:の Bulge_Erace(*l*) で *l* 回目の bulge 削除を行う．bulge 削除が完了した後 11:の Send_Message(*l*) で次スレッドに対応したフラグを更新することで一連のタスクが完了

したことを通知する．
この方式はフラグ確認のためにスピンロックを行い共有メモリに対して一定のコストがかかる排他処理を行い続ける．そのためフラグ確認による排他処理回数が多い場合低速になる可能性がある．

5.4 スリープ方式による重複防止

メッセージ方式ではスピンロックによるビジーウェイト行うため，フラグが確保出来ない場合，確保できるまで一定のコストがかかる排他処理を行い続けることで待機する．スリープ方式はフラグが実行可能状態ではない場合，スピンロックによるビジーウェイトを行う代わりにスリープを行い待機する方式である．メッセージ方式と異なる部分はこのフラグ確保のための待機処理だけであり，他の部分は同一である．同時実行の模式図も同様の図 13 で示される．

スリープ方式のアルゴリズムは Algorithm7 のように表現される．
重複防止方式としてはメッセージ方式とほぼ同様であるが，フラグを確認し実行可能状態ではないため待機する際，14:, 18:の Sleep のように, ループを行う代わりにスリープを行い待機する．先行スレッドは計算が進み，次の計算が可能になった際 5:, 11:の Send_Message(*l*, *thread_id*) で後続スレッドに対して通知を行い，通知を受け取った後続スレッドはスリープから復帰しフラグを確認し次の処理を行う．
スリープ方式は OpenMP を用いて実装することが難しいため pthread を用いて実装を行った．
フラグの同期の際は先行スレッドと後続スレッド二つのスレッド間で同期をとればよい．しかし，OpenMP では細かな同期処理が不可能であるため全てのスレッド間で同期を取る必要がある．pthread を用いる場合 2 スレッド間の同期を行うことが出来るため，スリープ方式ではフラグ確認，更新の際に 2 スレッド間の同期を使用した．

Algorithm 7 静的方式:スリープ方式の同期アルゴリズムを用いた実装

Parallel: { スレッド数分並列に実行する }

```
1:  i = thread_num
2:  while i < n − 2 do
3:    if Check_Flag(i) then
4:      Band_Reduction(i)
5:      Sent_Message(i)
6:      k = i + 1
7:      l = 1
8:      while k + l × nb + 1 < n do
9:        if Check_Flag(l) then
10:         Bulge_Erace(l)
11:         Send_Message(l)
12:         l = l + 1
13:       else
14:         Sleep()
15:       end if
16:     end while
17:   else
18:     Sleep()
19:   end if
20:   i = i + thread_num
21: end while
```

第 6 章　性能評価実験

6.1　実験環境

6.1.1　実験環境

これまでの章で述べた村田法のスレッド並列化について，複数の重複防止方式を用いて実装し性能を測定した．実験を行った主な環境は表 1 に記述した．一部実験は比較として表 2 に記述した環境でも実験を行った．CPU 以外の環境はほぼ同一である．

プログラムは全て C 言語を用いて記述し，行列，ベクトルは全て倍精度 double を使用した．行列の格納形式は Fortran と同じ列優先形式を使用し，行列, ベクトルに対する演算に関しては，行列-ベクトル積は dgemv, 行列-行列積は dgemm, rank-2 更新は dsyr2k を使用するなど，ほぼ全て BLAS ライブラリを使用して計算を行った．実行速度は帯幅縮小の計算量である $6nbN^2$ を実行時間で割ることで求めた．

表 1: 実行環境 (Intel 社 Core i7)

CPU	Intel Core i7 2600K 3.4GHz 4core(論理 8core) L1 Cache 32KB, L2 Cache 256KB, L3 Cache 8MB
Memory	8GB
OS	Fedora 18
Compiler	gcc 4.7.2
Compiler Option	-O3
BLAS	ATLAS-3.9.44
LAPACK	LAPACK-3.3.1

表 2: 実行環境 (AMD 社　 Phenom)

CPU	AMD Phenom II X4 960T 3.0GHz 4core L1 Cache 32KB, L2 Cache 512KB , L3 Cache 6MB
Memory	8GB
OS	Fedora 18
Compiler	gcc 4.7.2
Compiler Option	-O3
BLAS	ATLAS-3.9.44

6.1.2 格納形式

行列の格納形式として，行列のすべての部分を保持する全行列形式ではなく，計算に必要な部分のみを保持する対称帯行列形式を使用した．今回使用する行列は対称行列であり，演算後も対称性を失わないため，行列の上三角，もしくは下三角部分のみを保持すれば十分である．また帯行列は帯幅分の対角要素を除いてすべて 0 であるため，帯行列を保持するためには帯幅分の対角要素部分を保持すれば良い．村田法では帯幅縮小のための部分行列演算に伴う bulge 発生により帯外に半帯幅 nb 分の要素が発生する．発生した bulge は bulge 追跡により消去されるため nb 以上に拡大することはない．これらのことから村田法に使用する実対称帯行列の格納は図 14 のように圧縮を行うことが出来る．帯の要素を格納する対角要素と半帯幅分の $nb + 1$ と bulge を保持する nb，合計 $2nb + 1$ 行使用する．対称帯行列に変更することで，計算に使用する添字を変更する必要が発生する．図 14 の青色，緑色の計算に使用する部分行列は，全行列形式では長方形だが，対称帯行列格納形式では斜めの四角形となっている．しかし，次の列までの距離を示すリーディングディメンションをずらすことで，全行列形式の計算に使用した BLAS 関数をそのまま使用することが出来る．

全行列形式に必要な記憶領域が N^2 であるのに対して，今回の対称帯行列形式では $(2nb + 1)N$ でありメモリ容量を大幅に削減できる．また保持する行列サイズが小さくなることでキャッシュに乗りやすくなるため実行速度が向上する場合がある．

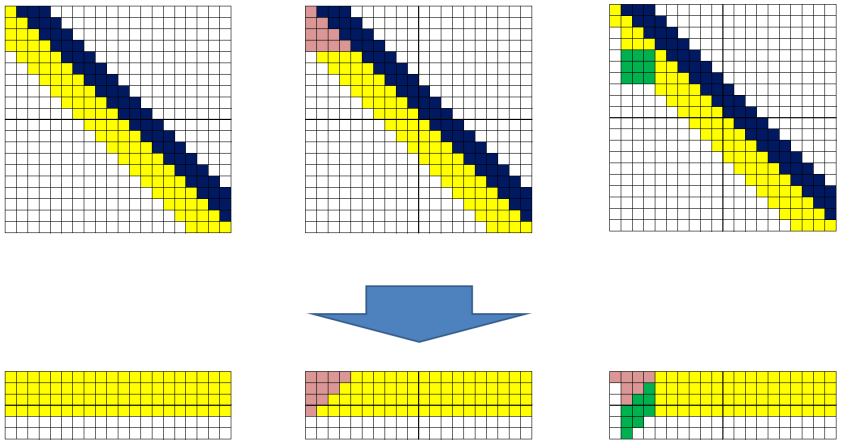

図 14: 全行列から対称帯行列への格納方法の変更

6.2 実験項目

今回の実験では，村田法の並列化による実行速度の変化の測定と分析を行うため以下の事柄について実験を行った．

- OpenMP と Pthread を使用した場合の排他処理実行時間の測定
  OpenMP と Pthread を使用してスレッド化を行った場合，それぞれ共有メモリに対する排他処理にかかる時間が異なる．それぞれの排他処理にかかる時間の測定を行った．
- 村田法のスレッド並列化による性能比較
  村田法の並列化による実行速度の変化を測定するため，各重複防止方式において，行列サイズを固定し帯幅とスレッド数を変化させた場合の実行速度の測定を行った．
- 同期方式による性能比較
  各重複防止方式の中で最も高速な方式を探索するため，行列サイズ，帯幅，スレッド数をそれぞれ変化させた場合の実行速度の測定を行った．
- キャッシュ効率による性能比較
  第 1 章に記述したように，キャッシュ効率は性能に影響を与えることが考えられる．PAPI[付録 A.1] を用いて，行列サイズを固定し，帯幅とスレッド数を変化させた時の L1 キャッシュ，L2 キャッシュ，L3 キャッシュのキャッシュミス回数について測定を行った．
- 同期回数による性能分析
  各重複防止方式によって同期処理を行う回数が大きく変化し，同期処理は全スレッド間で一貫性を保つためコストが大きいため，性能に影響することが考えられる．Scalasca[付録 A.2] を用いてスレッド数，重複防止方式を変化させた時の同期回数と実行時間の測定を行った．

6.3 OpenMP と Pthread を使用した場合の排他処理実行時間

6.3.1 実験内容

本論文では OpenMP と Pthread を使用してスレッド並列化を行う．この二つの方式では共有メモリ領域に対する排他処理とメモリ内容を一致させる同期処理にかかる時間 (以下この二つの処理を同期処理としてまとめて記述) が異なる．スレッド数を増やした場合同期処理にかかる時間が性能に影響することが考えられるため，本実験では表 1 と表 2 の各環境で OpenMP と Pthread をそれぞれ使用してスレッド並列化を行った場合の同期処理

にかかる時間を測定した．
同期処理の実行時間を測定するため，全スレッドで順番にフラグの確認と更新を行うプログラムを使用した．スレッドは循環リスト構造になっており，先行スレッドと後続スレッドを持ち，リストの先頭要素の先行スレッドはリストの最後尾である．スレッドは共有メモリ領域に存在するフラグを通してメッセージのやりとりを行う．各スレッドは共有メモリ領域に存在する自身に対応するフラグを確認し，先行スレッドがフラグを更新していれば後続スレッドのフラグを更新する．後続スレッドはフラグ更新を確認して，また後続スレッドのフラグを更新する．この処理だけを規定回数行った場合の実行時間を OpenMP と Pthread, それぞれを使用した場合について測定した．今回 100000 回処理を行った場合の実行時間の測定を行った．
先行スレッドによってフラグが更新されていない場合，OpenMP を使用する場合ビジーウェイトを行い先行スレッドによって更新されるまで問い合わせ続けて待機する．
Pthread を使用する場合スリープ方式を使用しフラグを確認した際，先行スレッドによって更新されてなければスリープを行う．

6.3.2 実験結果

図 15 が表 1 の環境で同期の実行時間のグラフ，図 16 が表 2 の環境での実行時間のグラフである．縦軸は実行時間 (s)，横軸はスレッド数である．
スレッド数が増えた場合同期処理にかかる時間は増加している．図 15 から表 1 の環境では OpenMP と Pthread1 で排他処理と同期処理にかかる時間には大きな差がないことがわかる．図 16 から表 2 の環境ではスレッド数が多い場合 OpenMP を使用した場合非常に時間がかかることがわかる．
表 2 の環境では OpenMP を使用した並列化に時間がかかるため，スレッド数が増加し同期処理が頻発することが考えられる場合には Pthread を使用した方が高速に計算できると予測される．

6.4 村田法のスレッド並列化による性能比較 (Core i7)

6.4.1 実験内容

村田法の並列化による実行速度の変化を見るため LAPACK による逐次帯幅縮小関数 DSBTRD と各排他制御方式において，スレッド数を変化させた場合の速度変化について実験を行った．各排他制御方式について，行列サイズを 10240 で固定し，帯幅を 32 から 288 まで 32 刻みで変更しながら実行速度を測定した．

Core i7 を使用した場合

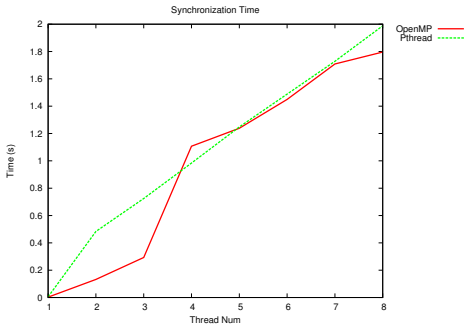

図 15: スレッド数を変更し 100000 回排他フラグ更新を行った実行時間 (s)

Phenom を使用した場合

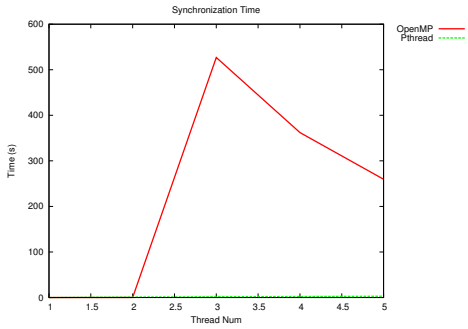

図 16: スレッド数を変更し 100000 回排他フラグ更新を行った実行時間 (s)

6.4.2 実験結果

図 17 が閉塞方式，図 18 がメッセージ方式，図 19 がスリープ方式の結果を表すグラフである．横軸は半帯幅 nb ，縦軸は実行速度 (flops) である．

全ての排他制御方式において，村田法を使用した帯幅縮小と LAPACK を使用した帯幅縮小を比較した場合，村田法を使用した方が高速に計算を行うことが出来ている．帯幅 96 で逐次実行した村田法と LAPACK を比較した場合 1.89 倍高速化できている．また帯幅によって高速化度合いが変化しているが，並列化することによって逐次方式よりも高速化することが出来ている．帯幅 96 で最も高速なメッセージ方式を使用した場合，逐次実行した場合より 3.93 倍高速化することが出来た．

帯幅が小さい時論理コア数と一致している 8 スレッドが高速である．物理コア数を超えた 5 スレッドの場合速度が低下しているが，スレッド数が 6, 7, 8 と増えるに連れて徐々に高速化し, 4 スレッドの場合よりも高速化している．論理コア数を超えた 9 スレッドになると急速に速度が低下している．論理コア数を超えた場合コアに割り当てられたスレッドを切替る処理が頻発するため，その影響ではないかと考えられる．今回の村田法による帯幅縮小計算は，物理コア数を超えて高速化できる. 一般に SMT は計算密度が十分に高い場合は有効性が低いが，計算量がそれほど大きくなく同期待ちによりマルチコアプロセッサの演算器が遊休になる場合には有効であると考えられる．そのため村田法帯幅が小さい場合は計算密度が高くないことが考えられる．したがってまだ改善の余地がある．

帯幅が大きくなるに従い実行速度は低下している．この低下速度はスレッド数が大きいほど急速に低下している．そのため今回の実験で最大の帯幅 288 の場合，速度低下が少ないスレッド数 2 が一番高速になっている．１スレッドの場合は速度低下が発生しておらず，またスレッド数が多い場合の速度低下は，１スレッドの実行速度に近づくと緩やかになっている．このことから帯幅拡大による速度低下は，1 スレッドの場合の速度に近づいてい

ると考えられる．今回の実験では帯幅の拡大を 300 程度で停止しているが，400, 500 程度になると並列化した場合でも性能が向上せず，逆にスレッド間の同期処理の分だけ遅くなる可能性がある．

各同期方式による特徴について見てみると，閉塞方式では帯幅の増加によるスレッド数が多い場合の性能低下が早く発生し，また急激に低下している．

メッセージ方式は閉塞方式と比較して性能低下の発生が遅い．スリープ方式では性能低下の発生は閉塞方式と同じ程度に始まるが，性能低下の速度は閉塞方式と比べて穏やかである．どの同期方式においても 1 スレッドの場合の速度はほぼ同じなので，帯幅が 400, 500 と十分に大きくなった場合同期方式による実行時間の差は発生しなくなる可能性がある．

閉塞方式を除き，帯幅が小さい場合論理コア数と最速のスレッド数が一致していた．したがってよりコア数の多いプロセッサを使用することで，スレッド数を増やし高速に計算ができるようになると考えられる．しかし，現在のプロセッサのコア数では 32, 64 コア程度で上限になってしまう．より並列性を高めるためには MPI 並列化を行い，複数のプロセッサと分散メモリを用いて計算を行う必要がある．現在のスレッドによる並列化方式はメモリを共有していることを前提にしているため, MPI によるメモリを共有しないプロセスによる並列化にそのまま適用することはできない．並列化方式をプロセスにまたがった bulge 追跡を行い MPI が適用できるアルゴリズムに変更する必要がある．

閉塞方式アルゴリズム (Core i7)

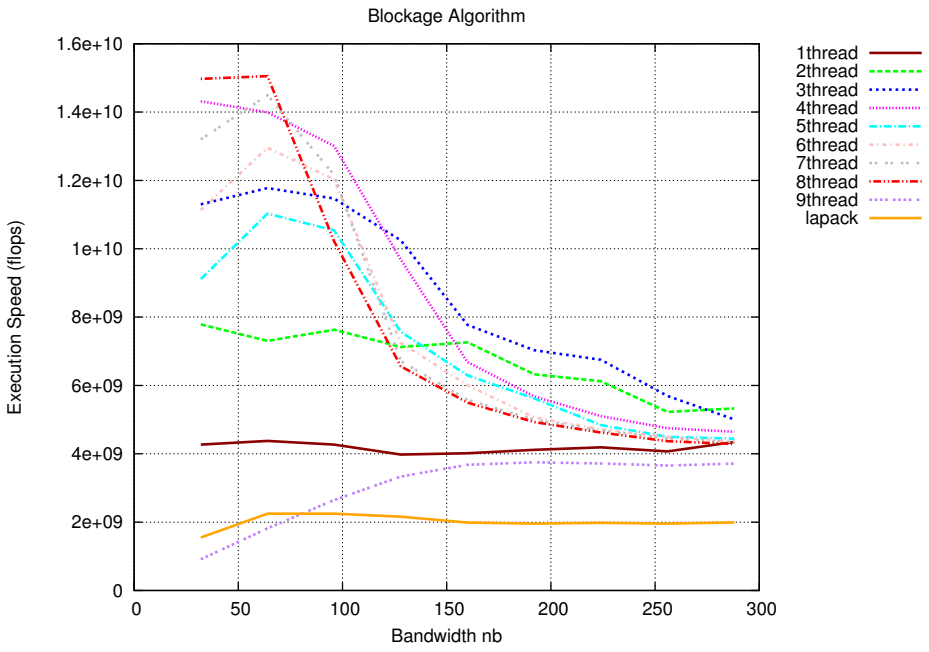

図 17: 帯幅，スレッド数変更時の実行速度 (flops)

メッセージ方式アルゴリズム (Core i7)

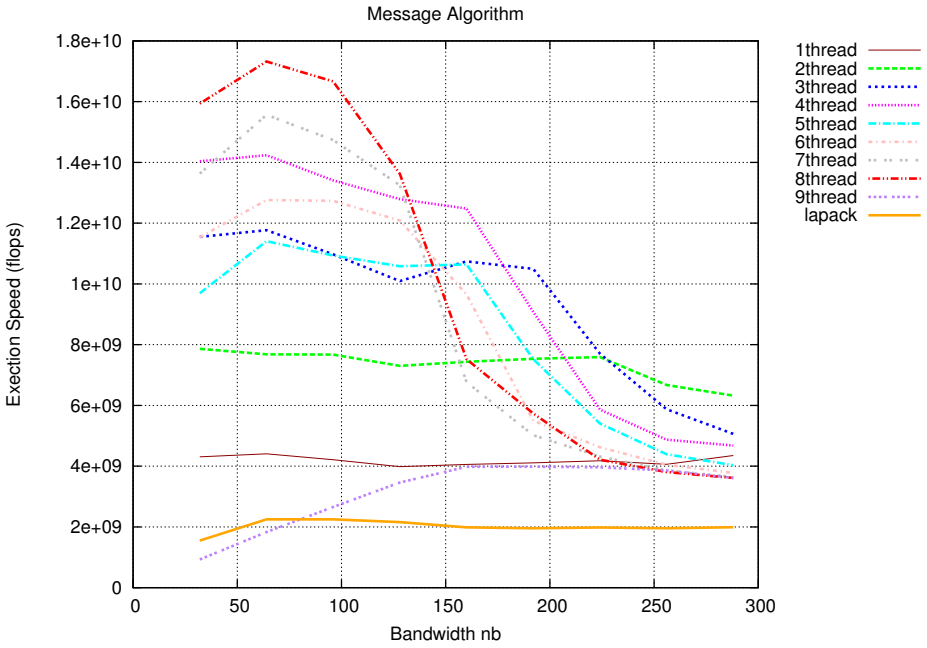

図 18: 帯幅，スレッド数変更時の実行速度 (flops)

スリープ方式アルゴリズム (Core i7)

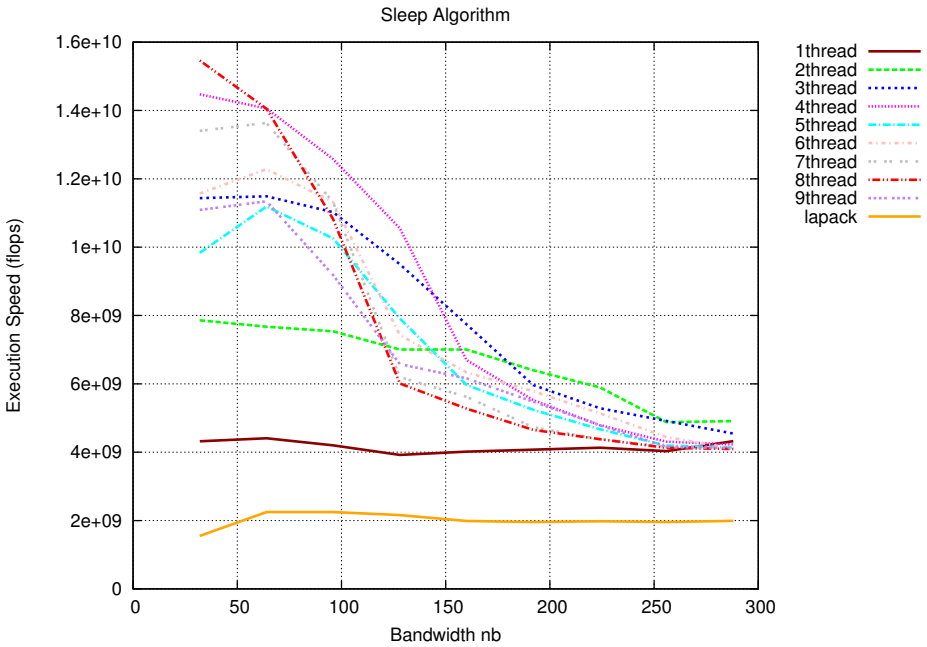

図 19: 帯幅，スレッド数変更時の実行速度 (flops)

6.5 村田法のスレッド並列化による性能比較 (Phenom)

6.5.1 実験内容

比較のため，表 1 に加えて表 2 の環境でも閉塞方式，メッセージ方式，スリープ方式の 3 方式での実行速度の測定を行った．行列サイズは 10240 に固定し，帯幅は 32 から 288 まで 32 刻みで変更しながら測定を行った．Phenom の物理コア数，論理コア数はともには 4 であるためスレッド数は 1, 2, 3, 4, 5, 8 の場合の測定を行った．

6.5.2 実験結果

図 20 が閉塞方式，図 21 がメッセージ方式，図 22 がスリープ方式の結果を表すグラフである．横軸は半帯幅 *nb*, 縦軸は実行速度 (flops) である．

メッセージ方式，スリープ方式の場合コア数に一致している 4 スレッドの場合が一番高速であるが，閉塞方式の場合 4 スレッドではなく 3 スレッドが高速である．5 スレッドの場合などスレッド数がコア数を超えた場合性能が低下するが，スリープ方式の場合性能の低下が小さい．これは同期にかかる時間が関係していることが考えられる．実験 6.3 から今回の環境では OpenMP を使用した場合スレッド数が増えると急激に同期処理に時間がかかっている．閉塞方式において高速なスレッド数とコア数が一致しない原因もこの同期処理時間の増加が関係していると考えられる．

帯幅が一定以上大きくなると計算速度が低下しているが，実験 6.4 と比較して速度低下が発生する帯幅が大きい．実験 6.4 と同様に帯幅増加による速度低下はスレッド数が大きいほど急激であり，帯幅が大きくなると 1 スレッドの場合と実行速度の差がなくなると予測される．

各同期方式による特徴を見てみると，閉塞方式の場合高速なスレッド数とコア数が一致しない．スレッド数が増えると急激に速度低下が発生している．これは閉塞方式は同期回数が多いため，時間のかかる同期処理が性能に影響を与えていることが原因だと考えられる．

メッセージ方式の場合高速なスレッド数はコア数と一致しており，スレッド数がコア数を超えると急激に速度が低下している．これも同期処理が原因と考えられる．

スリープ方式の場合高速なスレッド数はコア数と一致しており，スレッド数がコア数を超えても性能低下は穏やかである．これはスレッド数が増加しても一回の同期処理にかかる時間が急激に増加することがないことが原因だと考えられる．

本実験を行った表 2 の環境では OpenMP を使用した場合，スレッド数が増加すると急激に一回の同期処理に時間がかかる．このような環境ではコア数を増やしても性能の向上は難しい．Pthread を使用すると性能の低下が抑えられるため環境によって OpenMP と Pthread, 同期方式を使い分ける必要がある．

閉塞方式アルゴリズム (Phenom)

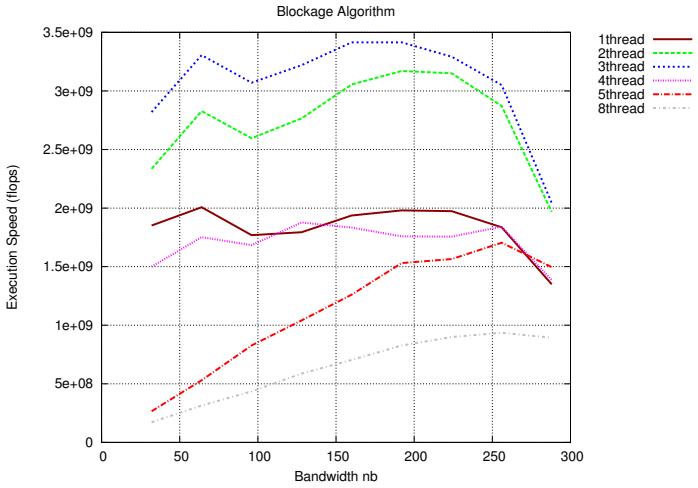

図 20: 帯幅，スレッド数変更時の実行速度 (flops)

メッセージ方式アルゴリズム (Phenom)

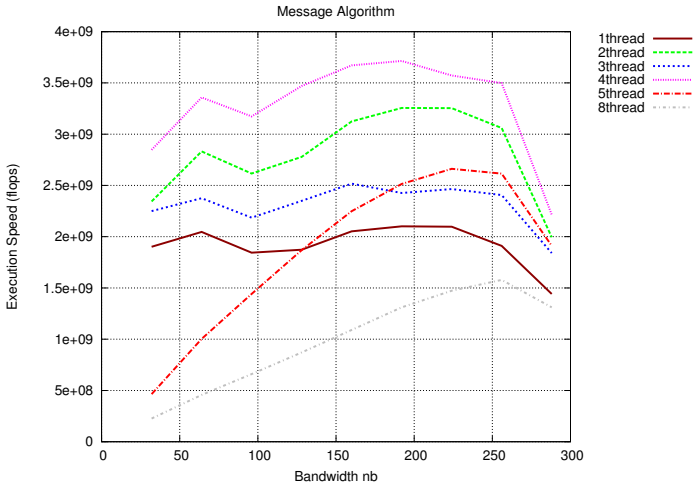

図 21: 帯幅，スレッド数変更時の実行速度 (flops)

6.6　3 方式で行列サイズ，帯幅を変更した際の性能変化 (Core i7)

3 種類の重複防止方式を用いて並列化を行った場合の実行速度について，行列サイズ，帯幅を変更しながら測定を行った．[16] から行列サイズが 5000 以下で帯幅 100 以下のの場合の結果を測定した結果，変化の途中で計測が終了してしまった．そのため今回行列サイズと帯幅を拡大し，行列サイズ 5120 から 19456 まで 1024 刻み，帯幅は 32 から 288 まで 32 刻みである．スレッド数は前述の実験 6.3 において，主要なスレッド数と考えられるスレッド数 2, 4, 8, 9 の場合の測定を行った．

スリープ方式アルゴリズム (Phenom)

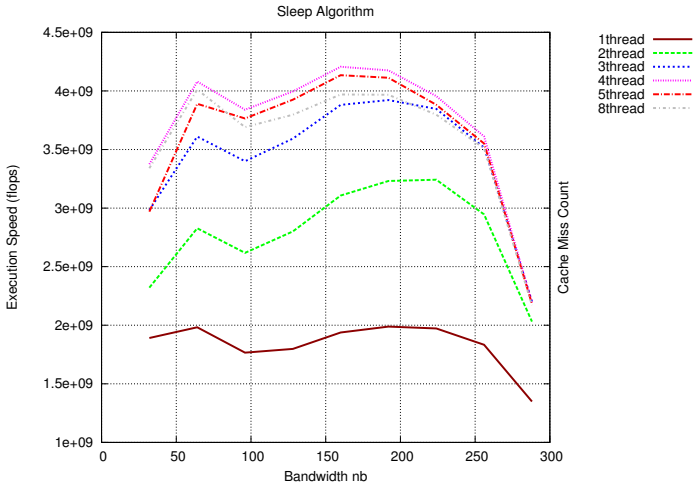

図 22: 帯幅，スレッド数変更時の実行速度 (flops)

6.6.1 実験結果

結果は図 23 である．右奥への横軸が帯幅 (*nb*)，左奥への縦軸が行列サイズ (*N*)，高さが実行速度 (flops) を表している．測定結果を見ると帯幅が小さい場合はスレッド数 8 のメッセージ方式，ある程度大きくなった場合スレッド数 4 のメッセージ方式，その後スレッド数 2 のメッセージ方式が高速である．

ほぼ全ての場合においてメッセージ方式が高速であったが，帯幅が小さい場合一部スリープ方式, 閉塞方式が高速の場合も存在している．特に閉塞方式の場合，今回の測定行列サイズ最大の場合において一番高速である．そのためより行列サイズを大きくした場合閉塞方式が高速になる可能性がある．

帯幅によって高速化の度合いが大きく変化している．最も高速なのは帯幅 64, 96 程度である．今回の実験では帯幅縮小操作のみを扱ったが，実際には帯行列化操作を組み合わせて使用されることが考えられる．帯幅縮小操作において高速な帯幅が帯行列化において低速な帯幅である場合，三重対角化全体としての実行速度が低下する可能性がある．帯行列化と合わせて高速化の度合いを見ることで，三重対角化操作全体で最適な帯幅を見ることが出来ると考えられる．

行列サイズを変更しても実行速度の変化はあまりない．しかし，行列サイズが小さい場合と，一部行列サイズにおいて高速化，もしくは低速化するサイズが存在している．帯幅を拡大した場合，並列化による高速化ができなくなる予測と合わせて行列サイズと帯幅を拡大した際の実行結果を測定するとより傾向が明確になると考えられる．

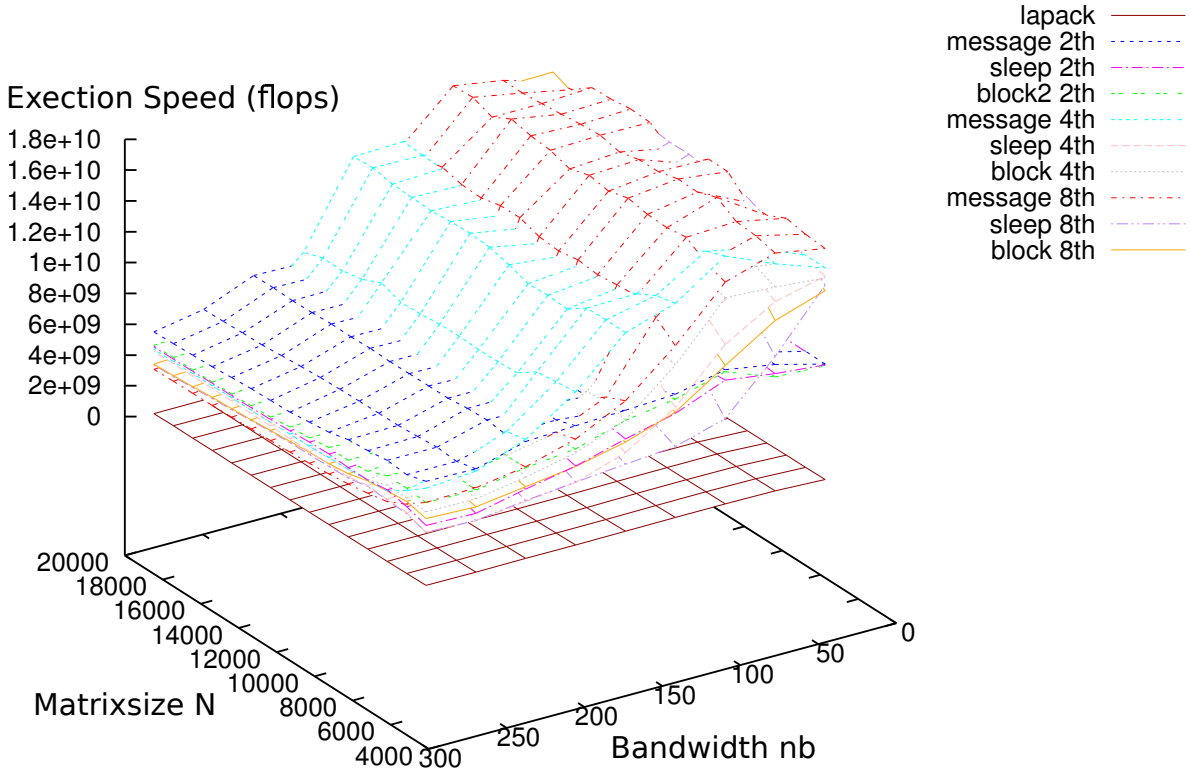

図 23: 3 種類の重複防止方式でスレッド数を変更した際の性能比較 (Core i7)

6.7 3 方式で行列サイズ，帯幅を変更した際の性能変化 (Phenom)

6.7.1 実験内容

実験 6.6 と同様に，3 方式の重複防止方式で並列化を行った場合の実行速度について，行列サイズ，帯幅を変更しながら測定を行った．表 2 の実験環境を使用し結果を比較する．表 2 の環境では測定にかかる時間が実験 6.6 の場合よりも大きいため，より小さな行列で測定を行った．行列サイズを 5120 から 10240 まで 1024 刻み，帯幅は 32 から 288 まで 32 刻みである．スレッド数は実験 6.5 の結果から，主要なスレッド数と考えられる 2, 4 の場合の測定を行った．

6.7.2 実験結果

実験結果は図 24 である．右奥への横軸が帯幅 (*nb*), 左奥への縦軸が行列サイズ (*N*), 高さが実行速度 (flops) を表している．

多くの場合において 4 スレッドのスリープ方式が高速である．行列サイズが小さく，帯幅が大きい場合一部 2 スレッドのメッセージ方式が高速である．スレッド数による性能低下の発生が緩やかなのでこの結果になったと考えられる．4 スレッドの閉塞方式は並列化しない場合よりも低速であった．本環境では OpenMP を使用した場合，同期に時間がかか

るため，高速に同期を行うことが出来る Pthread を使用したスリープ方式が高速になり，同期処理が頻発する閉塞方式が低速になったと考えられる．

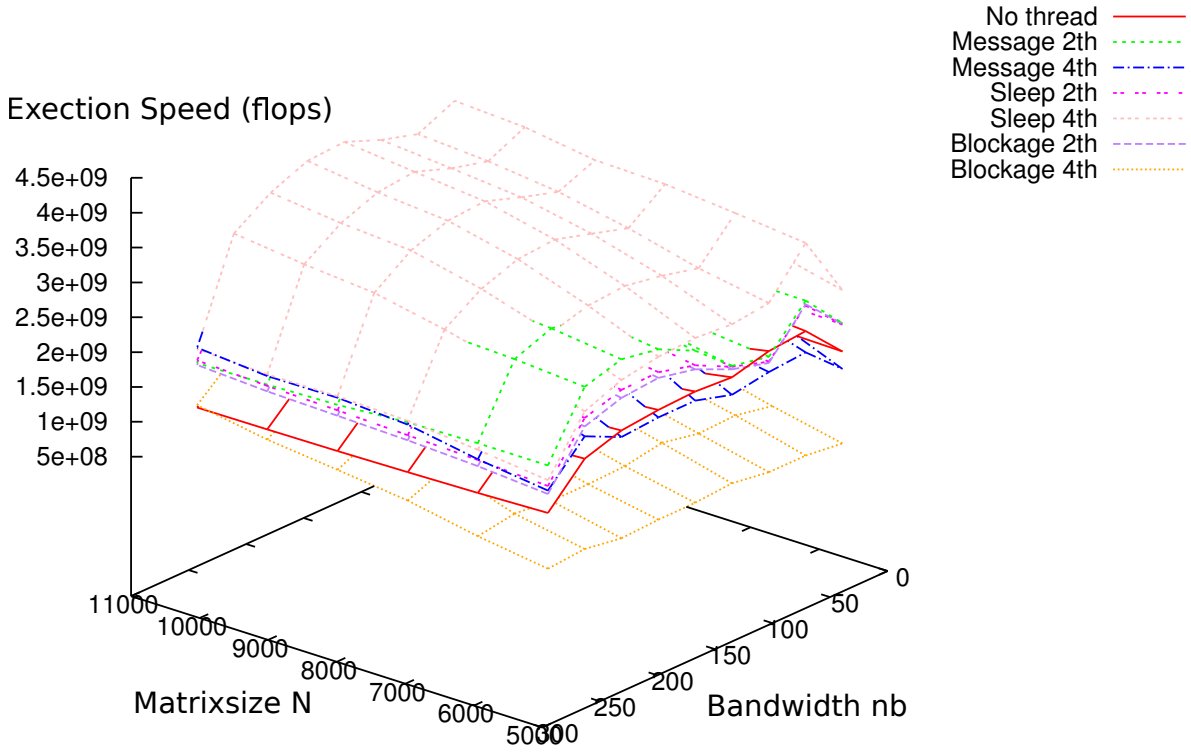

図 24: 3 種類の重複防止方式でスレッド数を変更した際の性能比較 (Phenom)

6.8 3 方式で帯幅，スレッド数を変化させた際のキャッシュミス回数の変化 (Core i7)

6.8.1 実験内容

実験 6.4 において，帯幅を変更した際のスレッド数別の実行速度を測定した場合，帯幅によって高速なスレッド数が変化した．その原因を考察するため，原因の一つとして考えられるキャッシュ効率について測定を行う．キャッシュ効率の指標としてキャッシュミス回数の測定を行う．各同期方式において行列サイズを固定し，帯幅を増加させた場合のスレッド数別の L1, L2, L3 キャシュミス回数の測定を行った．

6.8.2 実験結果

実験結果のグラフは図 25 ~ 36 である．行列サイズは 10240 に固定し，スレッド数は前節の実験 6.3 で変化が大きい，または重要だと思われる主要なスレッド数 1, 2, 4, 8, 9 を使用した．実行速度とキャッシュミス回数の関係を見やすくするため，各重複防止方式ごとのグラフには実行速度とキャッシュミス回数の両方を描写した．実行速度とキャッシュミ

ス回数を表す線において，スレッド数が同じ場合同じ色になっている．横軸は帯幅 (*nb*)，左縦軸は実行速度 (flops)，右縦軸はキャッシュミス回数である．

図 25, 図 26 で閉塞方式とメッセージ方式における L1 キャッシュミス回数を見てみると，帯幅が増加するにつれてキャッシュミス回数は増加している．帯幅が小さい場合スレッド数によるキャッシュミス回数に明確な差は存在しないが，帯幅が増加するとスレッド数 8 9 の場合のキャッシュミス回数はスレッド数 1, 2 の場合よりも増加した．閉塞方式では, 4 スレッドの場合キャッシュミス回数が増加しているが，メッセージ方式では 4 スレッドの場合においてもキャッシュミス回数は 1, 2 スレッドの場合とほとんど変化しなかった．

図 27 を見てみると，スリープ方式では帯幅の増加によるキャッシュミス回数の増加は発生しているが，スレッド数によるキャッシュミス回数の大きな差は発生していない．

3 方式のキャッシュミス回数をまとめた図 28 を見ると，スレッド数が多い場合の閉塞方式においてキャッシュミス回数が多いことがわかる．スレッド数が 1, 2 の場合，またスリープ方式，メッセージ方式の 4 スレッドの場合ではキャッシュミス回数にほとんど差はない．閉塞方式，メッセージ方式では実行速度の傾向とある程度類似性が見られるが，スリープ方式では一致しない．

図 29, 図 30 で閉塞方式とメッセージ方式の L2 キャッシュミス回数を見ると，L1 キャッシュミス回数と同様に，帯幅が増加するとキャッシュミス回数も増加し，スレッド数 8, 9 の場合特に増加した．両形式のスレッド数が 1, 2 の場合とメッセージ方式のスレッド数 4 の場合ではキャッシュミス回数は変化しない．

スリープ方式の L2 キャッシュミス回数 (図 31) を見ると，スリープ方式でスレッド数によるキャッシュミス回数の大きな差はあまり発生していない．スレッド数 8 の場合に一部増加しているが，帯幅が増加するとまた他スレッド数の場合と一致している．

3 方式のキャッシュミス回数のまとめ (図 32) を見ると，スレッド数が多い場合，閉塞方式においてキャッシュミス回数が多いことがわかる．スレッド数 1, 2 の場合，またスリープ方式，メッセージ方式の 4 スレッドの場合ではキャッシュミス回数に大きな差はない．閉塞方式，メッセージ方式では実行速度の傾向とキャッシュミス回数はある程度類似性が見られる．

図 33, 図 34, 図 35 を見てみると，L3 キャッシュミス回数はスレッド数が 1 の時がキャッシュミス回数が多いことや，スレッド数 9 よりもスレッド数 4 の方が多いこともあり，実行結果と密接な関係が見られれない．L3 キャッシュメモリは 8MB とサイズが非常に大きいため，今回の帯幅とスレッド数では，一回の帯幅縮小，bulge 消去に使用する全要素がキャッシュメモリ上に存在するためキャッシュミスが頻発しない可能性がある．

図 28,32 の L1, L2 キャッシュミス回数をみると閉塞方式，メッセージ方式実行速度とある程度関係していると考えられる．スリープ方式の場合は全スレッドの場合でほとんど変化しない．実行速度はキャッシュミス回数がある程度関係しているが，同期処理にかかる

時間がより影響している可能性がある．

閉塞方式重複防止アルゴリズム
(Core i7)

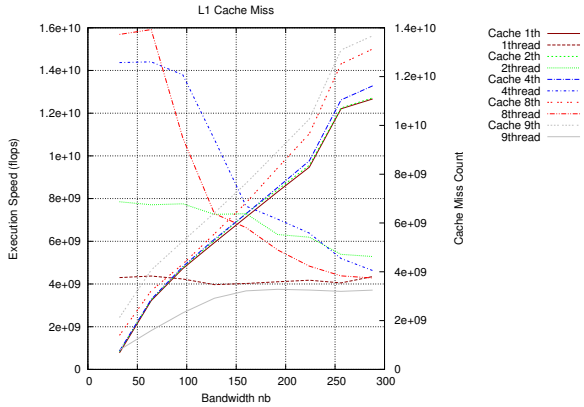

図 25: 帯幅，スレッド数変更時の
L1 キャッシュミス回数

メッセージ方式重複防止アルゴリズム
(Core i7)

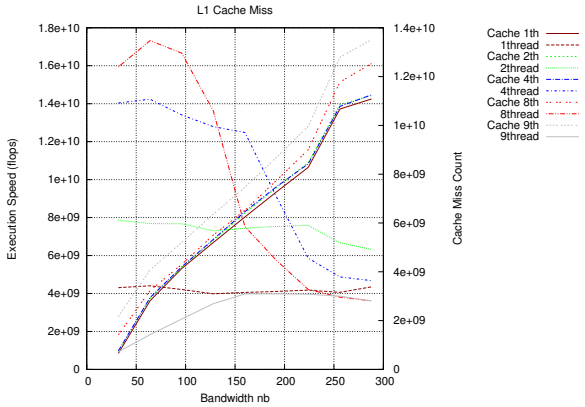

図 26: 帯幅，スレッド数変更時の
L1 キャッシュミス回数

スリープ方式重複防止アルゴリズム
(Core i7)

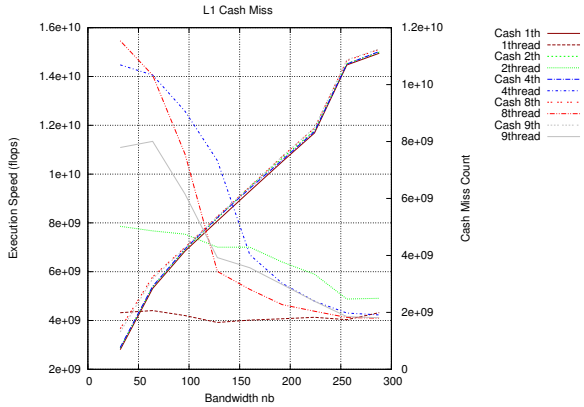

図 27: 帯幅，スレッド数変更時の
L1 キャッシュミス回数

3 方式の重複防止アルゴリズム
(Core i7)

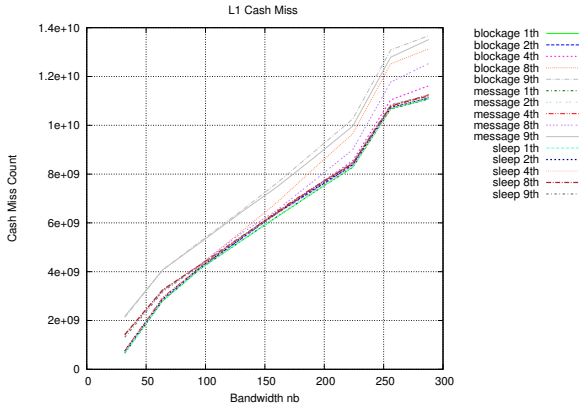

図 28: 帯幅，スレッド数変更時の
L1 キャッシュミス回数

6.9　3 方式で帯幅，スレッド数を変化させた際のキャッシュミス回数の変化 (Phenom)

6.9.1　実験内容

実験 6.5 においてスレッド数による実行時間の変化にキャッシュ効率が関係しているか調べるため表 2 の環境で測定を行う．キャッシュ効率の指標としてキャッシュミス回数を使

閉塞方式重複防止アルゴリズム
(Core i7)

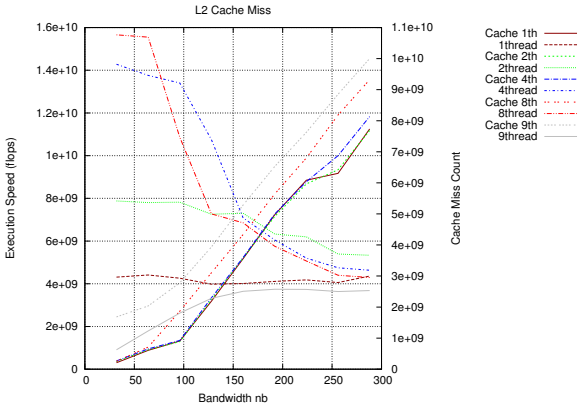

図 29: 帯幅，スレッド数変更時の
L2 キャッシュミス回数

メッセージ方式重複防止アルゴリズム
(Core i7)

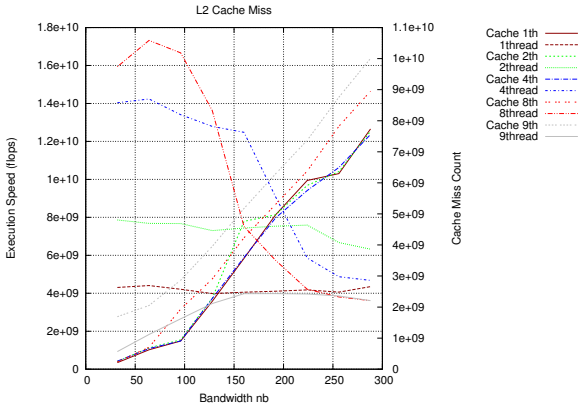

図 30: 帯幅，スレッド数変更時の
L2 キャッシュミス回数

スリープ方式重複防止アルゴリズム
(Core i7)

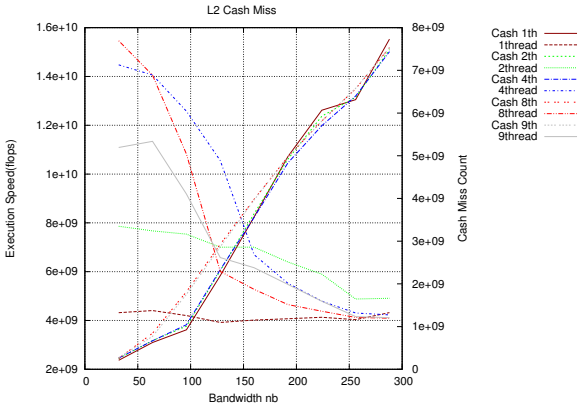

図 31: 帯幅，スレッド数変更時の
L2 キャッシュミス回数

3 方式の重複防止アルゴリズム
(Core i7)

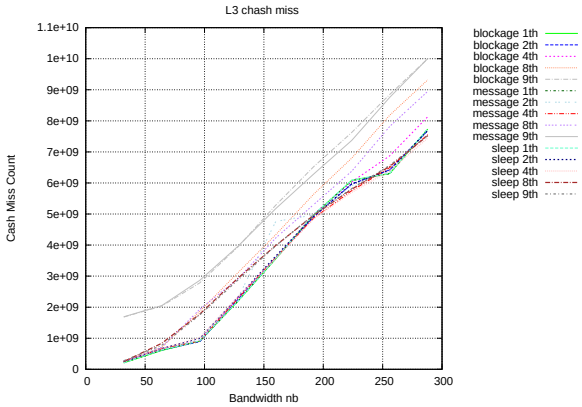

図 32: 帯幅，スレッド数変更時の
L2 キャッシュミス回数

用する．各同期方式について行列サイズを 10240 に固定し，帯幅を 32 から 288 まで 32 刻みで変化させて L1, L2 キャッシュミス回数の測定を行った．

6.9.2 実験結果

実験結果のグラフは図 37，38 である．行列サイズは 10240 に固定し，スレッド数は 1, 2, 3, 4, 5 を使用した．横軸は帯幅 (*nb*), 左縦軸は実行速度 (flops)，右縦軸はキャッシュミス回数である．

図 37, 38 を見ると，閉塞方式とメッセージ方式でスレッド数がコア数を超えた場合急激

閉塞方式重複防止アルゴリズム
(Core i7)

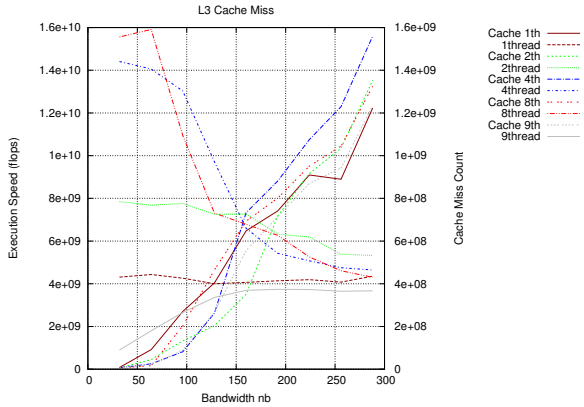

図 33: 帯幅，スレッド数変更時の
L3 キャッシュミス回数

メッセージ方式重複防止アルゴリズム
(Core i7)

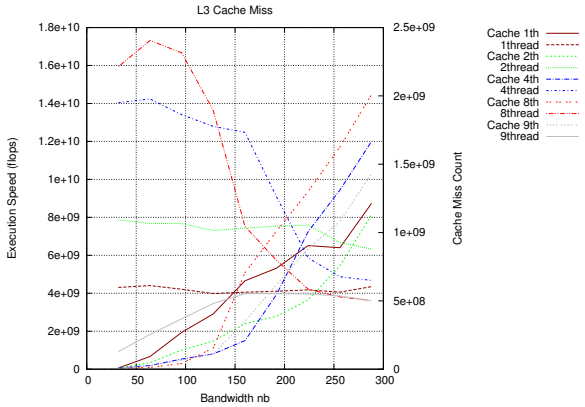

図 34: 帯幅，スレッド数変更時の
L3 キャッシュミス回数

スリープ方式重複防止アルゴリズム
(Core i7)

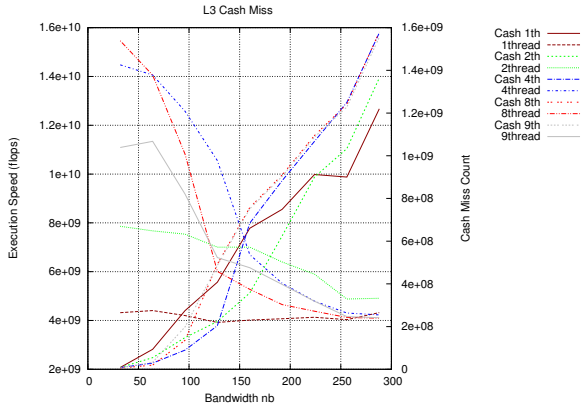

図 35: 帯幅，スレッド数変更時の
L3 キャッシュミス回数

3 方式の重複防止アルゴリズム
(Core i7)

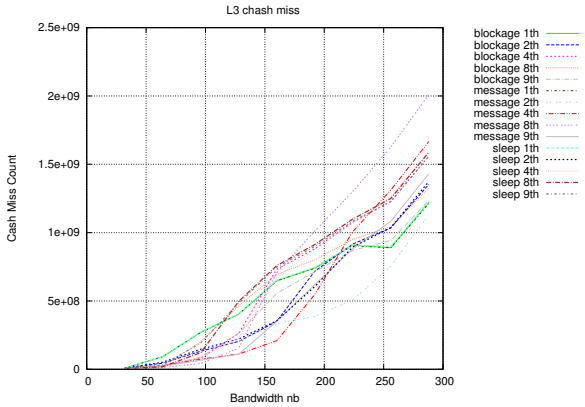

図 36: 帯幅，スレッド数変更時の
L3 キャッシュミス回数

にキャッシュミス回数が増加している．

閉塞方式はスレッド数の増加によるキャッシュミス回数の増加が顕著に発生しており，4 スレッドの場合に急激なキャッシュミス回数の増加が見られる．この傾向は実行速度の結果と一致している．メッセージ方式ではスレッド数が増加するとキャッシュミス回数は増加しているが，大幅な増加は見られない．スリープ方式はスレッド数が増加してもキャッシュミス回数の増加は発生しておらず，コア数を超えてもキャッシュミス回数の急激な増加は発生していない．

三方式において，閉塞方式を除いてスレッド数の増加による大幅なキャッシュミス回数の

増加は見られない．低速なスレッド数とキャッシュミス回数が多いスレッド数が一致する場合が存在することから実行速度の原因との一つとは考えられるが，主要な原因ではないと予測される．

3 方式の重複防止アルゴリズム (Phenom)

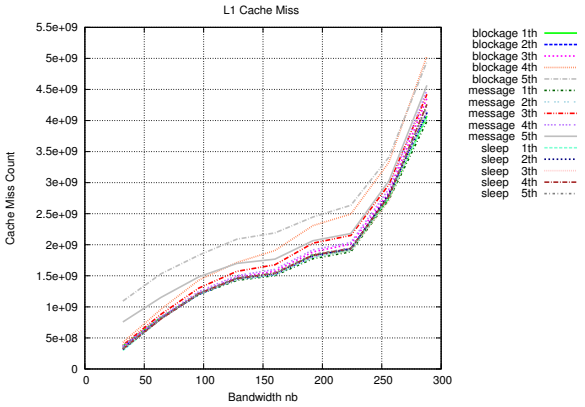

図 37: 帯幅，スレッド数変更時の L1 キャッシュミス回数

3 方式の重複防止アルゴリズム (Phenom)

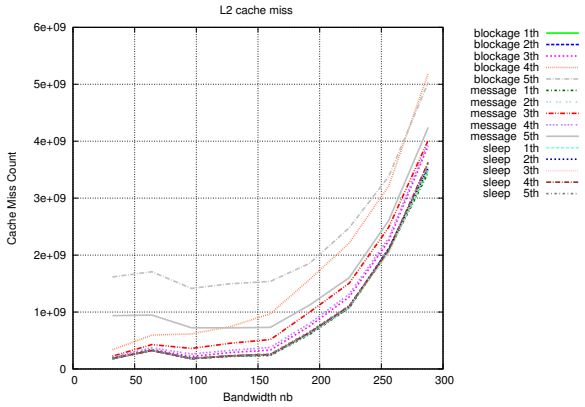

図 38: 帯幅，スレッド数変更時の L2 キャッシュミス回数

6.10 閉塞方式，メッセージ方式における同期回数の変化

6.10.1 実験内容

閉塞方式，メッセージ方式において，共有領域に対するフラグの参照やスレッド間での行列データの一貫性を保つための更新操作などの同期処理を行った同期回数と，処理全体の実行時間に対する同期処理の実行時間を測定した．

行列サイズは 10240 に固定し，帯幅を 64, 128, 256 に変更しながら，主要スレッド数である 1, 2, 4, 8, 9 の場合において同期処理を行った回数と実行時間を測定した．

スリープ形式はフラグが確保できない場合，ビジーウェイトではなくスリープを行い待機するため，同期回数は帯幅縮小，bulge 消去を行う回数の 3 倍程度であり，ほぼ固定であるため同期回数を測定しない．

Scalasca を使用することで，測定オーバーヘッドが発生するため Scalasca を用いない場合とは実行時間に差が生じている．そのため実際の同期回数や同期時間とは異なる可能性があるが参考値として同期回数と同期時間の測定を行った．

6.10.2 実験結果

同期回数は図 39 である．メッセージ方式では 9 スレッドの場合，同期処理が頻発しているが，1, 2, 4 スレッドの場合，帯幅が増加した場合も，同期回数の大幅な増加は見られ

ない．

閉塞方式では，スレッド数によって同期回数が大きく変化している．また帯幅が増加すると同期回数も大幅に増加している．しかし，スレッド数が増加した場合常に同期回数が増加しているわけではなく，スレッド数が小さい方が同期回数が多い場合もある．

同期回数の実行時間図 40 をみるとどちらの方式においても順当にスレッド数が増えると同期処理にかかる時間が増加している．Scalasca を用いた測定による測定オーバーヘッドが大きいため参考値ではあるが，スレッド数が増えると同期処理にかかる時間が増加する傾向は明確である．同期処理の回数と同期処理の実行時間はある程度の関係が見られる．しかし同期回数はスレッド数が小さいほうが多いが，同期時間はスレッド数が多い方が大きいことがあり，スレッド数が増加すると一度の同期処理にかかる時間が増大することがわかる．同期回数よりも一回の同期にかかる時間の方がより影響が大きいのではないか．

スリープ方式では同期処理回数は非常に少ないが，実行速度は高速化していない．スリープ方式におけるスレッドのスリープと起動はコストが大きく，ビジーウェイトを行った場合の数倍から数十倍低速である．これらの結果から，同期回数よりも，同期処理単体のオーバーヘッドがより大きな影響を与えることが予想される．

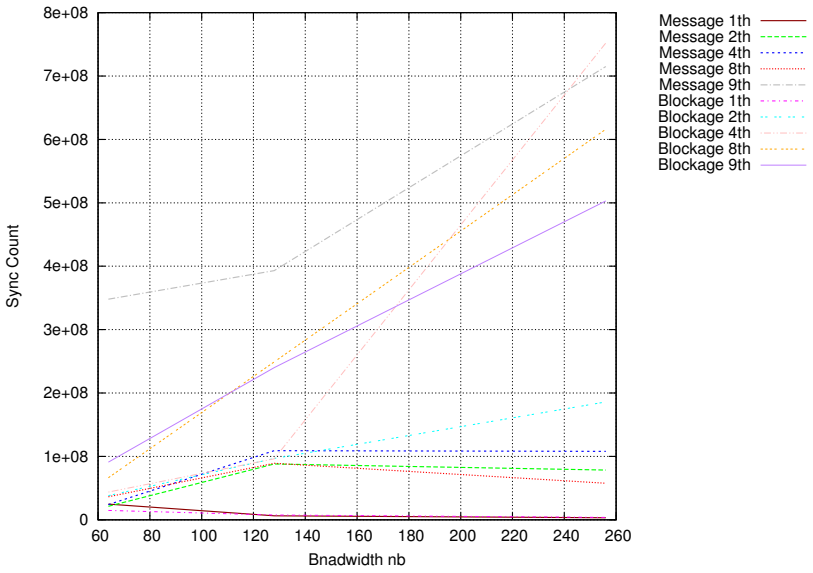

図 39: 閉塞方式．メッセージ方式を使用した際の同期回数

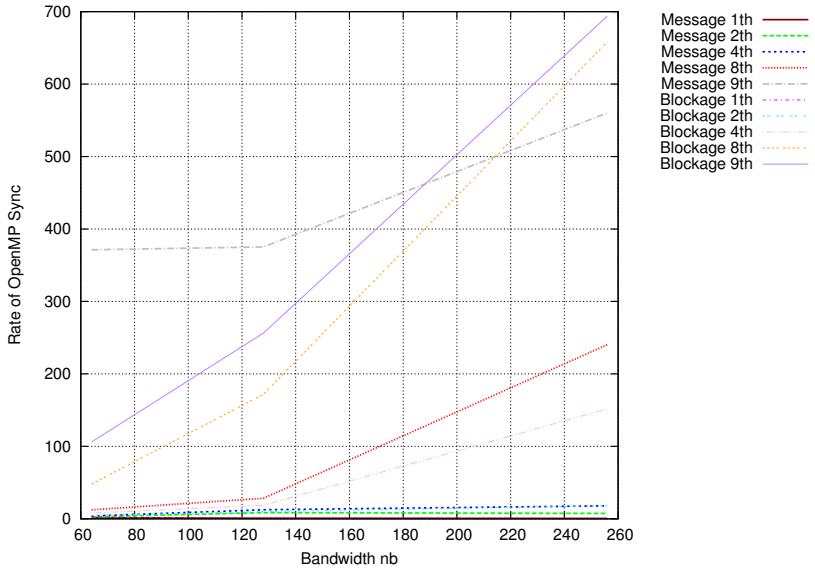

図 40: 閉塞方式．メッセージ方式を使用した際の同期処理時間

第 7 章　まとめ

今回の実験により，Intel Core i7 2600K プロセッサ (4core) 上においてベンチマーク行列として使用する行列サイズ 10240, 帯幅 96 において帯幅縮小操作に村田法を使用することで LAPACK による帯幅縮小操作より 1.89 倍高速化を行うことができた．さらに村田法のスレッド並列化を行うことで最も高速なメッセージ方式を用いて 8 スレッド並列実行を行った場合，1 スレッド逐次実行を行った場合よりも 3.93 倍高速化できた．

同期方式の違いにより高速なメッセージ方式と，低速な閉塞方式による 4 スレッド並列実行の時 1.03 倍の差が発生し，8 スレッド並列実行の時 1.59 倍の差が発生した．一部同期方式では特定のスレッド数で同期処理が頻発し低速化している．これらの結果から同期方式と実行速度に明確な相関があることが確認できるが．帯幅が増加した場合，同期方式による実行速度の差は減少している．

性能に影響する主な要因としてキャッシュミスによるペナルティと同期回数が考えれるため測定を行った．L1, L2 キャッシュミス回数からメッセージ方式，閉塞方式においてスレッド数の増加による速度低下と，キャッシュミス回数に多少の相関関係が確認されるが，スリープ方式については一致しない．そのため速度低下の主要な原因ではないと考えられる．

同期回数と同期処理の実行時間から同期回数はスレッド数が増加するにつれて増加するが一定ではなく，スレッド数が少ない場合にスレッド数が多い場合の同期回数を超える場合がある．しかし，同期時間はスレッド数が多い順になっているため，スレッド数が増加するにつれて一回の同期オーバーヘッドが増加していることがわかる．同期時間は一定の時間がかかり，スレッド数順になっていることからスレッド数の増加による速度低下の原因であると考えられるが，より定量的に関係を見るため追加実験を行う必要がある．

AMD Phenom 960 プロセッサ (4core) 上の結果と Intel Core i7 2600K プロセッサ (4core) 上の結果を比較した場合，異なる結果が得られた．Phenom を使用した環境ではメッセージ方式ではなくスリープ方式が高速である．これは OpenMP による同期時間が大きいことが影響していると考えられる．環境により同期時間と実行速度に大きな差が発生する可能性があるので各環境で確認する必要がある．

第 8 章　今後の課題

今後の課題として以下の事柄が考えられる．

- 異なる並列化方式との比較

  現在の並列化は，全スレッドが行列全体を保持している．もうひとつの並列化方式として図 10 のように行列の区間分割を行い，各スレッドは担当区間内の計算のみを行う．担当区間の端まで計算範囲をずらしながら計算を行い，端まで達した場合次の区間に必要な計算要素のみを渡すことで並列化を行うことができる．行列サイズが非常に大きい場合，こちらの並列化方式の方が，保持する行列が小さくなるため高速化できる可能性がある．現在の並列化方式と，この行列区間に分ける並列化方式のどちらが高速であるか分析を行う．
- MPI と併用 (ハイブリッド) することでより大規模な計算を行う．

  大規模計算を行うため MPI などを用いたプロセス並列化とスレッド並列化を併用したハイブリット形式について調査を行う． より大きな行列の計算を行う際，現在の 1 ノード内のスレッド並列化では限界がある．MPI などを使用したプロセス並列化を行い，大規模計算ができるようにする．その際には直前に述べた図 10 のように区間に分割する並列化方式を使用し，各プロセスに処理を割り振る．プロセス内部ではスレッド並列化を行うことで，大規模行列での高速計算ができる可能性があるため調査を行う．
- 調歩方式の実装

  同期方式として，調歩方式の実装を行い性能の比較を行う．調歩方式は，一回の帯幅縮小，bulge 消去の後に全スレッド間で同時に行うべき帯幅縮小，bulge 追跡が終了するまで待機する方法である．計算が最も遅いスレッドに合わせる必要があるが，同時実行性は最大になる．この方式を測定することによりスレッド処理の並列性が性能に与える影響を明確にすることができる．
- 帯行列化と合わせた性能調査

  今回の実験では帯幅縮小操作のみに注目して計算を行ったが，実際に使用する場合帯行列化と組み合わせて使用することが考えられる．最適な帯幅に関しても帯幅縮小だけを見れば最適な帯幅はかなり小さくなるが，帯行列化における最適な帯幅はどの程度か合わせて最適な帯幅を探索する．
- キャッシュラインの調整

  並列実行においてフラグ配列には非常に多くアクセスを行う．現在フラグは配列に並べているだけであるが，キャッシュラインに適切に乗るように調整を行うことで同期処理が高速化ができる可能性がある．この操作により一回の同期処理のコス

トを抑えることができる可能性があるため確認を行う.

付録 A　計測に使用したツール

A.1　PAPI

PAPI (Performance Application Programming Interface) は Tennessee 大学の ICL (Innovative Computing Laboratory) の Phil Mucci らが作成したハードウェアカウンタへのアクセスを行うための API である．

現在使用されているプロセッサの多くにはハードウェアカウンタと呼ばれるプロセッサ情報を記録したカウンタが存在する．ハードウェカウンタにアクセスすることで詳細なプロセッサ情報を取得することが可能になる．メーカーやプロセッサの種類によってハードウェアカウンタのへのアクセス方法は異なっているが，PAPI を使用することで，プロセッサによるアクセス方法の違いを意識することなくハードウェアカウンタにアクセスすることが可能になる．

PAPI を用いてハードウェアカウンタにアクセスすることで，キャッシュヒット率，キャッシュミス率，FLOPS, 読み込み，書き込み命令数などプロセッサに関する詳細な情報を取得することが出来る．これらのキャッシュヒット率や FLOPS などの値はイベントと呼ばれ, イベントをカウンタに対して設定することで，イベントで指定されたプロセッサに関する情報がカウンタに記録される．処理実行後にカウンタを確認することでイベントに設定された数値を取得することが出来る．

A.2　Scalasca

Scalasca は Forschungszentrum Jülich Jülich Supercomputing Centre と German Research School for Simulation Sciences Laboratory for Parallel Programming が共同で開発した並列アプリケーション向け性能分析ツールである．並列アプリケーションの動作を分析し，通信や同期処理を中心とした速度のボトルネックとなっている箇所，処理に時間がかかっている箇所を特定することが出来る．

MPI/OpenMP などに対応し各プロセス，スレッドごとの実行時間，通信時間，回数，関数ごとの実行時間など詳細な分析を行うことが可能である．

付録 B　SBR 法の並列化

B.1　SBR 法

村田法は Householder 変換を用いて一度に一行一列ずつ帯幅縮小を行うが，この計算は行列-ベクトル積の Level 2 BALS を中心に構成されている．Bischof らは一度に複数列の帯幅縮小をまとめて行うように変更し，データ再利用効率の良い行列-行列積を用いて計算を行う Succesive Band Reduction Algorithm(以下 SBR 法) を提案した [17]．図 41 に示すように一度に帯行列を三重対角行列に変換するのではなく，帯幅がより小さい行列に変換する操作を繰り返し，徐々に帯幅を縮小することで三重対角化を行う．

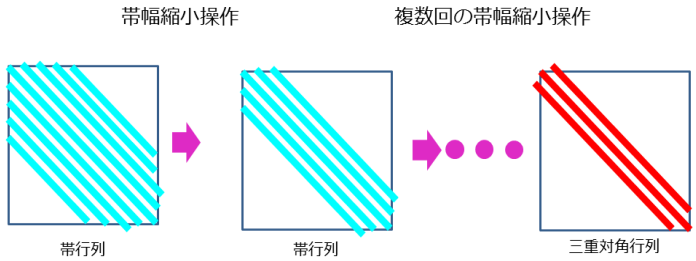

図 41: SBR 法による多段階帯幅縮小の概要

SBR 法では各反復での演算は，行列に作用する QR, PRE, SYM, POST の 4 つの操作と変換行列を作成する WY Representation の操作から構成される．各操作では次のような演算を行う．変換を行う前の半帯幅を b, 一度の変換で削除する行数を d, 一度の変換で帯幅縮小を行う列数，行数を nb, 変換行列が作用する部分行列の行数が h，変換後の帯幅を $\overline{b}$ とする．この時 $1 \leq d < b$，$nb \leq \overline{b}$ である．

- QR は帯行列の中の削除する部分行列に対して Householder QR 分解を行い上三角行列に変換する．その際使用した鏡像変換ベクトル u を並べた行列を U とする．QR 分解を行う部分行列のサイズは $h \times nb$ である．
- WY Representation は QR で使用した鏡像変換ベクトルを並べた行列 U から，変換行列 Q を作成する．変換行列 Q は一回行列を作用させることで，QR で行った複数回の Householder 変換と同じ作用をもたらす行列である ($M^{(nb)}...M^{(1)}A = QA$). $nb = 1$ の時 Q は QR で使用した Householder 変換行列である．
- PRE は QR 分解を行わなかった対角行列との間の部分行列に対して左から変換行列 Q を作用させる． PRE の行列サイズは $h \times (d - nb)$ である．
- SYM は行列の対角部分に対して左右から変換行列 Q を作用させる． SYM の行列

サイズは $h \times h$ である．

- POST は対角部分に変換行列 Q を左右から作用させたことで新たに発生する帯外の bulge を計算する．bulge 部分に対しては右側から変換行列 Q を作用させる．POST の行列サイズは $b \times h$ である．

図 42 は SBR 法による帯幅縮小操作の計算過程を表したものである．色の付いている部分が非ゼロ要素を表し，黄色の部分が QR, 水色の部分が PRE, 桃色の部分が SYM, 緑色の部分が POST を表している．村田法と同様に，帯幅縮小操作によって生じた bulge を右下まで追跡することで消去してから次の段の帯幅縮小操作を行っている．

$d = b - 1$，$\overline{b} = 1$，$nb = 1$ のとき SBR 法は村田法と同じ操作になる．

Algorithm 8 は上記の操作を行う SBR 法のアルゴリズムである．このアルゴリズムは帯幅縮小を一回行い帯幅 b から帯幅 $\overline{b}$ に変換するアルゴリズムである．帯幅を更に縮小するためには SBR 法による帯幅縮小を行った後，帯幅を $b \leftarrow \overline{b}$ として，nb, d, h の値を再度設定し SBR 法を同様に適用する．この操作を繰り返すことで徐々に帯幅を縮小する．

Algorithm 8 SBR 法のアルゴリズム

$\overline{b} = b - d$
for $j = 1$ to $n - \overline{b} - 1$ step nb **do**
  $j_1 = j;\ j_2 = j_1 + nb - 1;\ i_1 = j + \overline{b};\ i_2 = min(j + b + nb - 1, b);$
  while $i_1 < n$ **do**
    QR: $A_{[i_1:i_2, j_1:j_2]}$ を Householder QR 分解．上三角行列に変換する
    WY Representation : QR で使用した鏡像変換ベクトルから WY Representation を行いまとめて変換を行う行列 Q を作成する．
    PRE: $A_{[i_1:i_2, i_1:i_2]}$ に対して左から Q^T を作用させる．
      $A \leftarrow Q^T A$
    SYM: 対角部分 $A_{[i_1:i_2, i_1:i_2]}$ に対して左から Q^T 右から Q を作用させる．
      $A \leftarrow Q^T A Q$
    POST: $A_{[i_2+1:min(i_1+b,n), i_1:i_2]}$ に対して右から Q を作用させて bulge を作成する．
      $A \leftarrow AQ$
    $j_1 = i_1;\ j_2 = j_1 + nb - 1;\ i_1 = i_1 + b;\ i_2 = min(i_2 + b, n);$
  end while
end for

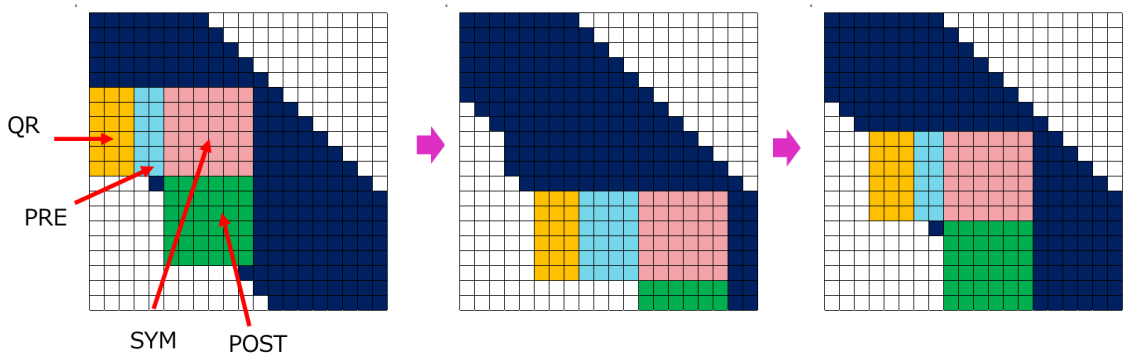

図 42: SBR 法の計算過程

B.2 並列化

SBR 法は図 43 のように計算範囲が重ならないため，村田法と同様に並列化を行うことが可能である．村田法と同様に，閉塞方式，メッセージ方式，スリープ方式を用いて並列化を行った．

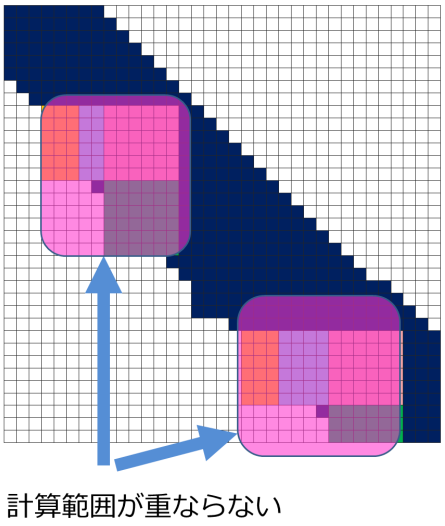

図 43: SBR 法の並列化

B.3 性能評価実験

B.3.1 実験環境

村田法と同様に表 1, 表 2 の実行環境で性能比較を行った．格納方式，行列サイズも同様である．

B.3.2 実験内容

閉塞方式，メッセージ方式，スリープ方式で SBR 法の並列化を行い実行速度を測定した．行列サイズは表 1 の環境の場合 5120 から 19456 まで 1024 刻み，表 2 の環境の場合 5120 から 10240 まで 1024 刻みである．帯幅は簡単のため 64, 128, 256 と 2 の累乗を使用する．一回の変換で行列サイズを 1/2 に縮小し，帯幅 32 まで縮小した後，村田法を用いて三重対角行列への変換を行った．村田法を使用する理由は，帯幅が一定以上小さい場合 SBR 法を使用して帯幅を縮小するよりも，村田法を使用して三重対角行列に変換した方が高速なためである．SBR 法で $d = b - 1$ として計算を行うと村田法と同じ計算になる．帯幅 64 の場合 64, 32 と帯幅を縮小し，その後村田法を用いて三重対角行列に変換を行う．帯幅 256 の場合 256, 128, 64, 32 と帯幅を 1/2 ずつ縮小し，その後村田法を用いて三重対角行列への変換を行った．

B.3.3 実行結果

図 44 が表 1 の環境での実行結果，図 45 が表 2 の環境での実行結果である．右奥への横軸が帯幅 (nb)，左奥への縦軸が行列サイズ (N)，高さが実行速度 (flops) を表している．

図 44 から，SBR 法は村田法を使用した場合よりも高速であり，スリープ方式を使用した場合が一番高速であることがわかる．多くの場合スレッド数 9 のスリープ方式が一番高速であるが，行列サイズが小さい場合スレッド数 8 のスリープ方式が高速である．行列サイズが小さく，帯幅が大きい場合一部 4 スレッドのメッセージ方式が高速である．

図 45 から表 2 の環境では，SBR 法は村田法を使用した場合よりも高速であり，スリープ方式とメッセージ方式を使用した場合高速であることがわかる．帯幅が小さい場合，5 スレッドのスリープ方式が高速であるが，帯幅が大きい場合 4 スレッドのメッセージ方式が高速である．

両方の環境において村田法の場合と異なり論理コア数を超えたスレッド数でも高速化している．このことから，SBR 法の計算は同期待ちによる演算器の遊休が多く発生し，計算密度が高くないことが考えられる．したがってよりスレッド数を増やすことで高速化することが出来る可能性がある．

行列サイズが小さく帯幅が大きい場合メッセージ方式が高速であるのは村田法の部分が高速に計算できていることが原因だと考えられる．行列サイズが小さい場合，全体の実行時

間が小さいため村田法を使用する際の影響が強く出るためではないかと予測される.

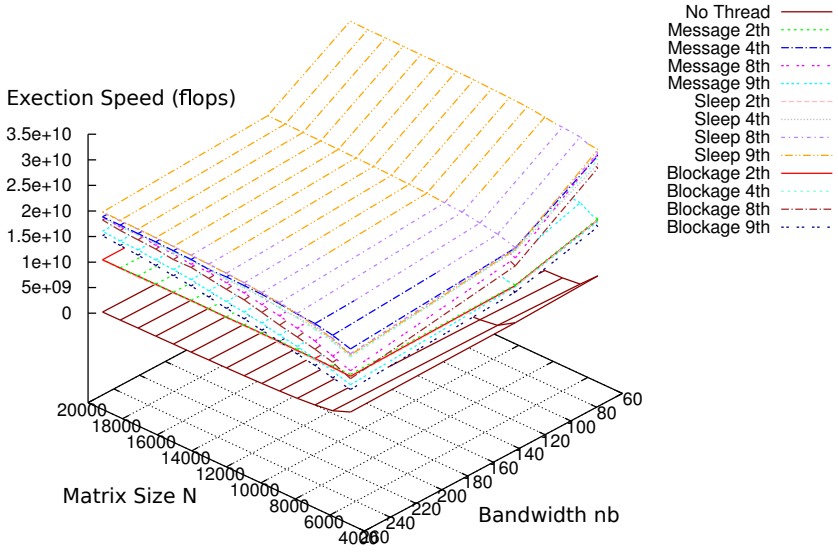

図 44: 3 種類の重複防止方式でスレッド数を変更した際の性能比較 (Core i7)

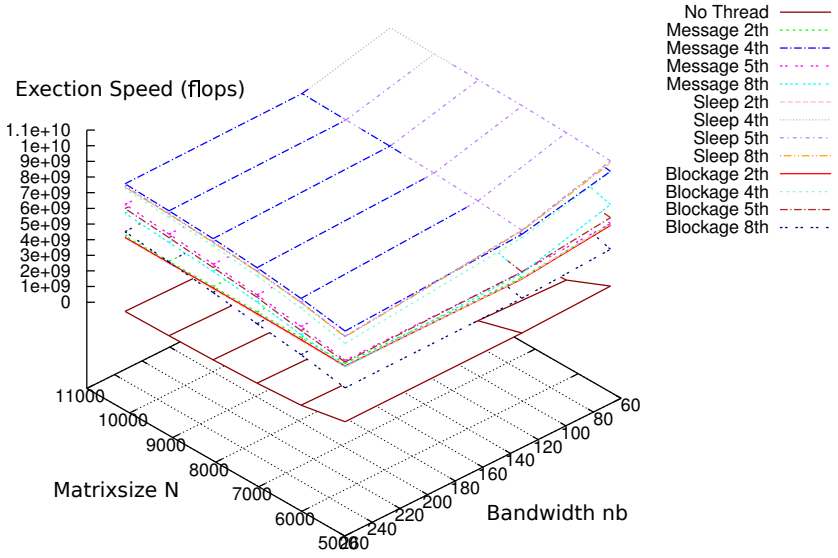

図 45: 3 種類の重複防止方式でスレッド数を変更した際の性能比較 (Phenom)

参考文献

[1] F. Chatelin, Valeurs propres de matrices, Masson, 1988. (F. シャトラン , 伊理正夫 , 伊理由美 (訳), 行列の固有値　新装版　最新の解法と応用, 丸善, 2003).

[2] Netlib, BLAS (Basic Linear Algebra Subprograms), University Of Tenesee , Netlib, 2005. http://www.netlib.org/blas/index.html.

[3] 今村俊幸, T2K スパコンにおける固有値ソルバの開発, スーパーコンピューティングニュース, Vol. 11, No. 6, pp. 1-21, November 2009. http://www.cc.u-tokyo.ac.jp/support/press/news/VOL11/No6/200911imamura.pdf.

[4] 戸川隼人, 新装版 UNIX ワークステーションによる科学技術計算ハンドブック, サイエンス社, 1998.

[5] J. J. Dongarra, S. J. Hammarling, D. C. Sorensen, Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations, Journal of Computational and Applied Mathematics, Vol. 27, No. 1-2, pp. 215-277, September 1989. http://www.netlib.org/utk/people/JackDongarra/PAPERS/031_1989_Block-Reduction-of-Matrices-to-Condensed-Forms-for-Eigenvalue-Computations.pdf.

[6] J. J. Dongarra, R. A. van de Geijn, Reduction to Condensed Form for the Eigenvalue Problem on Distributed Memory Architectures, Parallel Computing, Vol. 18, No. 9, pp. 973-982, September 1992.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK User's Guide Third Edition, SIAM, 1999. http://www.netlib.org/lapack/lug/.

[8] C. Bischof, B. Lang, X. Sun, Parallel Tridiagonalization through Two-Step Band Reduction, In Proceedings of the Scalable High-Performance Computing Conference, IEEE Computer Society Press, pp. 23-27, 1994.

[9] C. Bischof, C. van Loan, The WY Representation for Products of Householder Matrices, SIAM J. Sci. Stat. Comput, Vol. 8, No. 1, pp. 2-13, January 1987. http://dx.doi.org/10.1137/0908009.

[10] 村上 弘, ブロックハウスホルダ変換について (数値計算), 情報処理学会研究報告. [ハイパフォーマンスコンピューティング], Vol. 2005, No. 97, pp. 43-48, October 2005. http://ci.nii.ac.jp/naid/10016799404/.

[11] 村田健郎, 堀越清視, 対称帯行列を三重対角化するための新アルゴリズム, 情報処理, Vol. 16, No. 2, pp. 93-101, February 1975. http://ci.nii.ac.jp/naid/110002720181/.

[12] 小国力 (編), 村田健朗, 三好俊郎, J. J. Dongarra, 長谷川秀彦, 行列計算ソフトウェア WS スーパーコン 並列計算機, 丸善, 1991,

[13] H. Rutishauser, On Jacobi rotation patterns, Experimental Arithmetic, High Speed Computing and Mathematics, In Proceedings of Symposia in Applied Mathematics, Vol. 15, pp. 219-239, 1963.

[14] H. R. Schwarz, Algorithm 183: Reduction of a Symmetric Bandmatrix to Triple Diagonal Form, Commun. ACM, Vol. 6, No. 6 pp. 315-316, June 1963.

[15] L. Kaufman, Band Reduction Algorithms Revisited, ACM Transactions On Mathematical Software, Vol. 26, No. 4, pp. 551-567, December 2000.

[16] 白澤孝仁, 村田法のスレッド並列化によるマルチコア CPU 上での実対称行列帯行列帯幅縮小操作の高速化, ハイパフォーマンスコンピューティングと計算科学シンポジウム, pp. 09, January 2014.

[17] C.Bischof, B.Lang, X.Sun, A Framework for Symmetric Band Reduction, ACM Transcations On Mathematical Software, Vol. 26, No 4, pp. 581- 601, December 2000.

謝辞

本研究を進めるにあたり，熱心なご指導をいただきました理化学研究所計算科学研究機構の今村俊幸様，電気通信大学情報理工学研究科情報・通信工学専攻情報数理工学コースの岡本吉央准教授，緒方秀教教授に深く感謝の意を表します．
研究を進めるに当たり助言と刺激をいただきました，今村研究室，岡本研究室の皆様をはじめ多くの方々に手助けしていただきました．感謝の意を表します．